SCHEDULING JOBS IN THE GENERALISED JOB SHOP

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

ANKUR BHATNAGAR

to the

DEPARTMENT OF INDUSTRIAL & MANAGEMENT ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1996

CERTIFICATE

This is to certify that the present work on 'Scheduling Jobs in the Generalised Job Shop", by Ankur Bhatnagar has been carried out under my supervision and has not been submitted elsewhere for award of a degree.

(T. P. Bagchi)
Professor
Industrial & Management Engineering
Indian Institute of Technology
Kanpur- 208 016

16th April, 1996

16/4/96

CERTIFICATE

This is to certify that the present work on 'Scheduling Jobs in the Generalised Job Shop", by Ankur Bhatnagar has been carried out under my supervision and has not been submitted elsewhere for award of a degree.

(T. P. Bagchi)
Professor
Industrial & Management Engineering
Indian Institute of Technology
Kanpur- 208 016

16th April, 1996

16/4/96

ABSTRACT

The Standard Job Shop scheduling has always been a challenge to researchers. It is among the tougher problems in the class of *NP*-hard problems. Several approaches have been applied to it with the aim to develop efficient algorithms. Yet, existence of several methods notwithstanding, the standard job shop model is only of limited utility to practitioners because of its relatively narrow scope. A formulation and its Genetic Algorithm solution of the more general form of the job shop scheduling has been presented here. In the Generalised Job Shop, duplicate resources are available and an operation may require more than one resource of the same *type*. The operation may also require more than one type of resources — each in quantities of one or more units. Further, the precedence relationship among the operations need not be completely specified. The proposed model supports partial ordering of the operations i.e., some of the operations on a product may be performed in any order. This constraint may be extended so as to cover the general case of Open Shop scheduling problem. The products may not be available for processing simultaneously i.e., they may have different release dates. The resources also may not be available during some time intervals of the scheduling horizon due to various reasons like break downs, maintenance etc. The products may have different priorities among themselves. A methodical parameterisation of the genetic algorithm used for solving the generalised job shop scheduling problem has also been presented as an illustration. Such parameterisation of the genetic algorithms was first done by Krishan Raman. An exact algorithm has also been evolved by extending an algorithm provided by Balas for standard job shop. The extended method covers the constraint of more than one type of resources in each operation in addition to the constraints of standard job shop.

ACKNOWLEDGMENTS

First and foremost I express my heartfelt thanks to Dr. T. P. Bagchi whose guidance enabled me to venture into this topic. I am grateful for his valuable suggestions and excellent supervision throughout the span of this work.

I am also thankful to my senior, a Ph.D. student, Mr. Neeraj Kumar for his invaluable suggestions and help without which this thesis would have been in a different form. I wish to express my gratitude to Mr. Ajay Kansal for providing moral support to me during the course of my thesis work. I also wish to communicate my appreciation of Mr. Parijat Sahai's (M.Sc. Int. Maths, Final year) help, who like a true friend, made himself available whenever I needed his time, whether in typing the thesis or in discussing scheduling to get a better understanding of the concept. Last but not the least, I extend my thanks to all the Professors, my colleagues and the whole IME family who made my stay in IITK an enjoyable period.

April, 1996 Ankur Bhatnagar

TABLE OF CONTENTS

TABLE OF FIGURES

1. Generalised Job Shop Scheduling

Science and Technology have made breathtaking progress in the last few decades. Consequently, we have been continuously building, rebuilding and now reengineering various factories, industrial structures, institutions, systems of great size and complexity. We have problems of large scale and complexity on one hand and the society's increasing demand for reducing costs and speed of delivery through the channels of competition on the other. With such developments the need to economise and to make the systems more efficient and quick has been increasing faster and faster.

In such a scenario, Operations Research offers us a ray of hope and the tools for coping with these pressures. Over the last few decades Operations Research too has made significant progress. Newer and better tools have been developed and are still being developed for dealing with the problems of resource utilisation, timeliness etc. But before we can move ahead and take advantage of these tools, we must face a caveat — some of these tools are still not amenable to practical applicability.

On my part to exploit OR for any practical utility, a lot more is required than doing a few courses in them. Of course, this does not imply that theory does not have a significant role to play. On the contrary, a practice oriented operations researcher

must also have very strong theoretical foundation; but, a lot more. Stating it concisely, an extensive experience in modeling and implementing the OR solutions is required.

This thesis work is a step in that direction. Through this thesis, I hope to have grasped 'real issues' involved in such work by the end of it. I wish to develop an appreciation of the hurdles in modeling and implementations of solutions to the real life problems and the limits on efficacy of such solutions.

1.1 Standard job shop scheduling problem

There are p products to be made using q different machines. Each of the product has to pass through a fixed sequence of machines for processing. Problem is finding the order, for each machine, of the products to be processed on that machine so that the maximum completion time of all the jobs (makespan) is minimised. [1] [2] [3]

1.2 The generalised job shop scheduling problem

A generalised job shop (GJS) [4] may present the following picture:

At any given time there are some products occupying the resources in various stages of processing. Each *product* has a fixed sequence of processing on the resources in terms of *operations* (i.e., steps in processing). Each operation may require more than one resource at a time, e.g., an operation may be occupying one press, one die, one tool one jig, two operators of one skill and three operators of another etc.

The products possibly have different priorities. Each product has a different release date.

So even though the GJS problem appears to be akin to the standard job shop scheduling problem at first sight, it is actually more complex. The only similarity is that here also the products have a fixed sequence of operations through the resources (machines and others) and that the maximum utilisation of the resources is desirable (among other objectives, such as timeliness) which is achieved in the job shop scheduling.

The differences between the standard job shop and the generaslised job shop are as follows:

1. Rolling horizon

Unlike the standard job shop scheduling here there is no finishing line. The new orders keep coming and completed orders are dispatched when done.

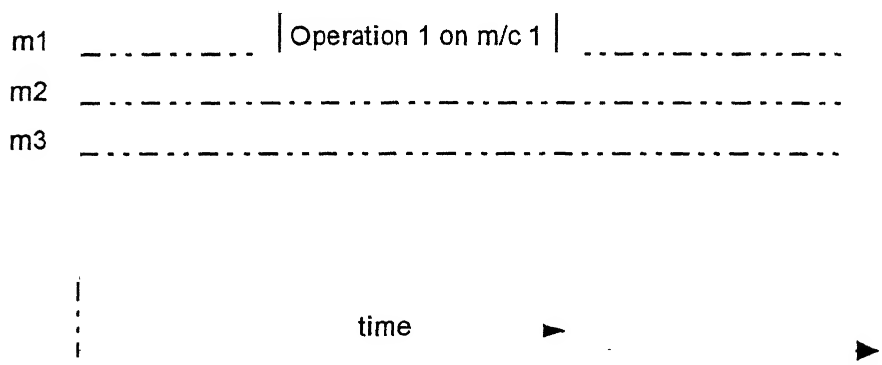
2. Availability of multiple resources

The generalised job shop has more than one machine of the same kind available. This is true with other resources also, especially rigs and operators. For example, we may

have 50 operators who can man the presses; we may have 5 200-ton presses; 10 lathe machines etc.

3. Different resources in an operation, more than one resource of each type

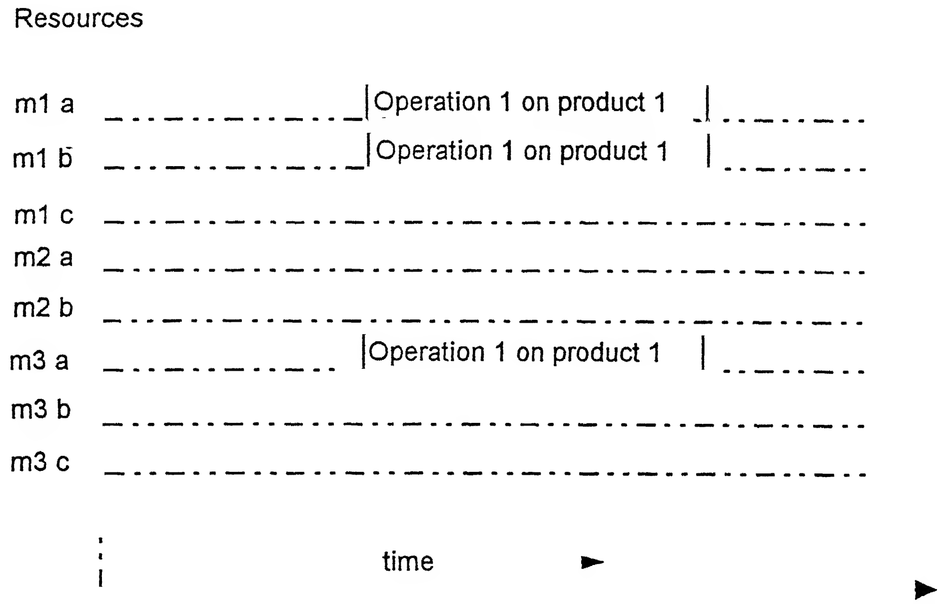
In the simple job shop scheduling there is only one machine required in each of the operation but in the GJS case the machine is viewed as a part of a broader perspective



Standard Job Shop: Each operation requires exactly one *type* of resource, exactly one unit of it; only one resource of a kind available.

Figure 1-1 A representation of the Standard Job Shop.

as a resource. A *resource* is defined as any facility which is used in an operation, and not just machines. For example, in a press shop apart from the presses, dies, jigs, tools, etc. are also resources. Even the operators are resources — human resources as they are often recognised. An operation may require more than one resource at the same time, for example, an operation may require a press, a die, a tool and an operator which may also be required in other operations. In the general case, an operation may require more than one resources of each type, e.g., more than one operator, more than one die, more than one jig.



GJS: Multiple resources, different resources in an operation, more than one resource of each type. Here operation 1 on product 1 requires two m1 type machines and one m3 type machine.

Figure 1-2 A representation of the Generalised Job Shop.

4. Priorities

A concept of priorities exists in GJS. Different products may have different priorities. Thus a performance measure will have to be evolved which would take care of the relative importance of the jobs as well as some time based optimisation criteria.

5. Optimisation criteria [3]

In the GJS we have to do optimisation in a rolling horizon. In such a situation the concept of makespan is changed. Looking back at the scheduling history of a shop, makespan alone cannot be defined as the measure of the performance. But whenever we are doing scheduling for the jobs at hand, we can define the makespan as the

maximum completion time of the jobs *at hand.* Using makespan as the objective function has the advantage that except in certain conditions, it results in a good utilisation of the resources. However makespan does not give any consideration to the relative importance of different products.

Having weighted completion time as the objective function circumvents this disadvantage. The scheduling is done keeping the priority of each product in mind. The weighted completion time is defined as follows:

$$WCT = \sum_{i=1}^{N} w_i C_i$$

where w_i is the weight (or priority) of the ith product and C_i is the completion time of the ith product, N is the number of products.

The scheduling history too therefore may be evaluated using weighted completion time. If the priorities are in proportion of the costs of the products, in-process inventory may be minimised.

There are several other optimising criteria which can be used for scheduling. Some of them are [3]:

- Maximum latenass (L_{max}).
- Discounted total weighted completion time ($\Sigma w_j(1-e^{-rC_j})$)
- Total weighted tardiness ($\Sigma w_j T_j$)
- Weighted number of tardy jobs ($\Sigma w_j U_j$)

6. Assembly type operations

This means that products may be interdependent. Some products are components to a sub assembly which in turn are fitted in another assembly. We can also have a concept of product groups as a direct extension of this.

By product group we mean that there can be a set of products which must be completed together even though they may not form a physical whole ultimately.

7. Release dates

In the standard job shop scheduling problem it is assumed that all the products are available for processing at the same time while actually, this may not be so in the generalised job shop. Apart from obvious constraints (e.g., the customer would provide the raw material for processing only after a specific date) which necessitate having release date, there is one more use of the release dates. At any point in time management has a set of assured products (for which order has already been received) to be processed (with different release dates) at hand. But management may also foresee the receipt of another order for which it might like to plan in advance. Thus the concept of release dates permits management to plan its activities thus being better able to optimise the various objectives.

8. Non-availability of resources

Resources may not be available during certain intervals. The reasons could be many, such as follows:

(i) Scheduled preventive maintenance of machines, unexpected breakdowns

(ii) Operators on leave or absent

(iii) Holidays

(iv) Strikes

(v) Non-availability of parts, spares etc.

(vi) Quality problems

9. Partial ordering of the operations

The operations need not be all related with each other through strict precedence constraints. For example, only *some* of the operations may be related with each other through precedence constraints on a product while others may be carried out in any order.

With *partial ordering*, the solution for GJS can also address the general problem of Open shop scheduling.

1.3 Mathematical formulation of the generalised job shop scheduling problem

The mathematical formulation presented here parallels that given by E. Balas [4] with some modifications. These modifications of the Balas formulation make it general enough to encompass above additional features that characterise the generalised job shop scheduling problem.

The formulation is in the form of integer programming, as follows:

Input variables:

N = set of all operations/activities.

n = number of operations.

t_i = starting time of the ith operation, $i \in N$.

c_i = processing time of the ith operation, $i \in N$.

R = set of operations having release dates (generally the first operation on each product.)

r_i = release date of the ith operation, $i \in R$.

P = set of operations having priorities attached with them. (Generally the last operations on the products.)

p_i = priority of the ith operation, $i \in P$.

Res = set of all types of resources.

$b_i(k)$ = Units of the ith type of resources available in time interval k, $i \in Res$.

a_{ij} = units of the ith type of resources required by the jth operation, $i \in Res; j \in N$.

Decision variables:

x_j^k = this variable can have only two values: (0,1). If it is 1 it means that operation j is being processed in time interval k.

1.3.1 Objective functions

The two commonly used performance measures of the GJS are:

1. Makespan, M.
2. Weighted Completion Time, W.

The objective function for makespan is:

Min $M = max_{i \in N} \{ t_i + c_i \}$

Note however that this function does not consider the priorities p_i of the operations – neither explicitly in the objective function, nor implicitly in the constraints. This is incorporated in the objective based on the weighted completion time.

The objective function for the weighted completion time is:

Min $W = \Sigma_{i \in P} \{ p_i (t_i + c_i) \}$

1.3.2 The constraints

$t_j - t_i \geq c_i \; \forall \; (i, j) \in H$ where $H \subset N \times N$, a matrix indicating the precedence relationship between jobs i and j.

$t_i \geq r_i \; \forall \; i \in R$ Release date constraints.

$t_i \geq 0 \; \forall \; i \in N - R$

$$\sum_{j \in J_i} a_{ij} x_j^k \leq b_i \quad \forall \; i \in Res, \; k = 1, 2, 3, \ldots\ldots., M.$$

J_i is the set of all operations requiring the ith resource. k is the time interval tag.

$$x_j^k = \begin{cases} 1 & t_j \leq k \leq t_j + c_j \\ 0 & otherwise \end{cases} \qquad \forall \; j \in N, \; k = 1, 2, 3, \ldots\ldots, M.$$

The above problem is a strongly *NP* hard problem. In even a small sized problem with 10 operations and 15 time periods, we will have 150 (0, 1) variables plus some more to explicate the relationship between x_j^k and t_j variables.

On the other hand, our present work indicates that using a heuristic search approach based on genetic algorithms one is able to tackle such problems within reasonable limits of time. For instance, as shown in this thesis, a problem involving 99 operations on 31 products, 41 machines to seek the minimum makespan schedule took about 60 seconds on the HP-9000 (850) system with the code executing in C.

2. Literature Survey

2.1 General overview

The job shop scheduling problem is a well-known and old scheduling problem. Numerous efforts have been made by a number of researchers in solving this problem using a variety of tools. One primary reason that has attracted the attention of researchers towards this problem is the practical benefits which would accrue from its solution. The job shop scheduling problem is one of those problems which are commonly encountered in day to day life of managers in diversly different settings. The need of such scheduling may arise in executing civil engineering projects, in running the production shops or anywhere else where the resources need to be allocated to activities. Phillips [5] has documented a case on a job shop with scheduling problems. In this case he reports how the proper scheduling of such a shop could result in major productivity gains and the consequent turnaround.

A very efficient heuristic used widely for solving the simple job shop scheduling problem is the Shifting Bottleneck Algorithm provided by Adams, Balas and Zawack [2]. In this algorithm the machines are scheduled one by one using the solution of $1|r_j|L_{max}$ problem (single machine, release dates, maximum lateness scheduling problem). Each unscheduled machine is evaluated for how much bottlenecking it causes. This is done by ignoring the

constraints imposed by other unscheduled machines and then scheduling the bottlenecking machine for $1|r_j|L_{max}$. The machine with the largest L_{max} is considered a bottleneck and then it is scheduled according to its solution. All the scheduled machines are again examined and resequenced using $1|r_j|L_{max}$. This process is repeated until all the unscheduled jobs are scheduled.

However, it is difficult to extend this method to cover the additional features of the generalised job shop scheduling problem, especially when multiple resources are required in the same operation. This is so because in the case of the simple job shop, each time a machine is scheduled, all operations to be done on that machine are scheduled simultaneously. This is not always possible in GJS.

Recently the job shop scheduling problem has been attacked on various fronts though these methods do not seem to tackle the problem in its full entirety. Some of the recent approaches are as follows.

Several investigators have presented new models, concepts and methods for solving the simple job shop scheduling problem. Notable among these are Otolorin [6], Alli [7], Van Laarhoven, Emile and Lenstra [8] and Applegate and Cook [9]. Otolorin has provided a method that restructures the job shop problem into an equivalent imaginary flow shop problem which is easier to solve mathematically. After applying an appropriate flow shop scheduling technique, the result is converted back to job shop problem. Alli solves the

problem by heuristically prioritising the jobs at different stages in scheduling. Van Laarhoven, Emile and Lenstra have proposed a method using the simulated annealing heuristic. Applegate and Cook have designed a new heuristic procedure for finding schedules, a cutting plane method for obtaining the lower bounds and a combinatorial branch and bound algorithm. Combining the heuristic and the branch and bound algorithm, they were able to solve the well-known 10×10 problem of J.F. Muth and G.L. Thompson in a relatively short computer time.

Some researchers have used knowledge-based or expert systems for tackling the problem of job shop scheduling. These systems solve the problems using their stored 'experience' and artificial intelligence. Beigel and Wink [10]; Odesseus Charalambus and Hindi [11]; Thomas Tsukada and Kung Shin [12]; Collinot & Le Pape [13] are the ones who have contributed in this field in the recent past. Beigel and Wink have developed a concept for an expert system that uses heuristics for setting priorities and for establishing sequencing during the early part of the schedule.

Simulation has also been put to use for solving the problem. Gaindhar, Pandey and Mishra [14] did a simulation study of the job shop scheduling. They also included the effect of machine failures in their study. They have used the simulation model to compute the optimum number of repairmen required for a particular set of machines. In reality, however, simulation does not evolve any schedule directly, rather it is an effective tool for *evaluating* a given schedule in an environment characterised with uncertainty. Therefore

one would recommend simulation as complementary to any scheduling technique. A good description of the simulation methodology is provided by Law and Kelton [38], including its application in the job shop.

It is quite possible that a good schedule given by a scheduling technique might not give good results when subjected to uncertainty in processing times, arrival times of jobs, breakdown times etc. but at the same time it is an *expectation* that a schedule which gives good performance under uncertainty will give good results under certainty also. Because of this reason, Genetic Algorithms score an advantage over other methods. Unlike other methods, GA provides a *set* of good schedules instead of one good schedule. Therefore, if one schedule does not prove acceptable on simulation, another good schedule found in GA can be tested.

The features of the Generalised Job Shop considered in this thesis project are: availability of multiple resources, each operation using one or more than one resource, one or more than one units of each type of resource, release dates, priority, partial ordering and non-availability of resources. There have also been many attempts towards solving the generalised job shop scheduling problem. Researchers have come up with newer and more ingenious ways to tackle the general features, however, none of them seem to have solved the problem with all its features. Nasir and Elsayed [15] have invented a method which takes into account multiple machines. It falls short of the true generalisation because, among other reasons, it does not solve the problem if the operations require more

than one unit of the *same* resource or more than one *type* of resources. A mixed integer programming formulation has been developed and then on this basis a new algorithm has been evolved. The authors have developed one additional algorithm based on SPT rule and have compared the two. Koulamas [16] has taken in account machine breakdowns and has developed an analytical procedure for determining the mean and standard deviation of job flow-time in a job shop. An expression for the optimal work in process inventory level is also developed for a job shop with machines subjected to random breakdowns or with two classes of jobs. Widmer [17] has developed a taboo search approach for solving the job shop scheduling problem with tooling constraints. On the other hand, Brucker and Schlie [18] have come up with a method to address the situation when an operation can be processed on any (one) of the machines from a specific set of machines.

Genetic Algorithms (GA) are essentially the summary of nature's process of evolution of life on earth. Over a short period of three billion years, the earliest living organisms which were far simpler and poor in adapting to the environment have developed into much more versatile, complex and robust organisms. This evolution has happened due to natural selection as proposed by Darwin. Genetic algorithms emulate this process of evolution to evolve better solutions of a problem from poor solutions. The genetic algorithms were invented by Holland in 1975 [31]. For an interesting overview of the GA, reader is referred to [32].

Many researchers have used GAs for solving the standard job shop scheduling problem. Nakāno and Yamada [33] have used binary representation for representing a schedule. They represent the schedule in terms of priority of a job in a pair of jobs on each of the common machines they will require for processing. They use binary digits to show the priority e.g., if *priority* $(o_{15}, o_{25}) = 1$ then it means that operation on job 1 will be processed before operation on job 2 on machine 5; otherwise the priority will be 0. Therefore for each job-pair, there is a sequence of binary digits showing priority. A complete schedule will be shown by all possible ${}^{N}C_2$ job-pairs (N = Number of jobs). In case a particular representation results in an illegal schedule, it is replaced with another which is closest to it and legal. Falkenauer and Bouffouix [19] have given another genetic algorithm approach. For simple job shop scheduling Biegel and Davern [20] also have provided yet another approach using GA. They have specifically addressed the task of generating inputs from the shop to the GA process for schedule optimisation. They have tried to integrate GA in a Computer Integrated Manufacturing (CIM) cycle. However, their method also solves only the simple job shop scheduling problem. Davis [35] has considered simple jobshop and suggested coded of the genotype (the representstion of feasible schedule in string format in GA) in terms of a "preference list" linked to each of the work stations. However, Davis' paper does not provide sufficient details of the representation of the problem.

We have additionally explored extension of an implicit enumeration algorithm given by Egon Balas [1] towards solving the GJS analytically. This method, after certain

modifications has been able to take into account the possibility of having more than one *type* of resource in an operation, but each resource existing only as one single unit. This method is an extension of Balas' approach [1] and is explained below in detail.

2.2 Balas' original enumeration approach to scheduling in the standard job shop

The following method of representing and solving the standard job shop scheduling problem was given by Egon Balas. He proved it that the solution generated through this method will always yield optimal results. For details we refer to his original paper: "Machine Sequencing Via Disjunctive Graphs Using an Implicit Enumeration Algorithm" [1]. The method proceeds as follows.

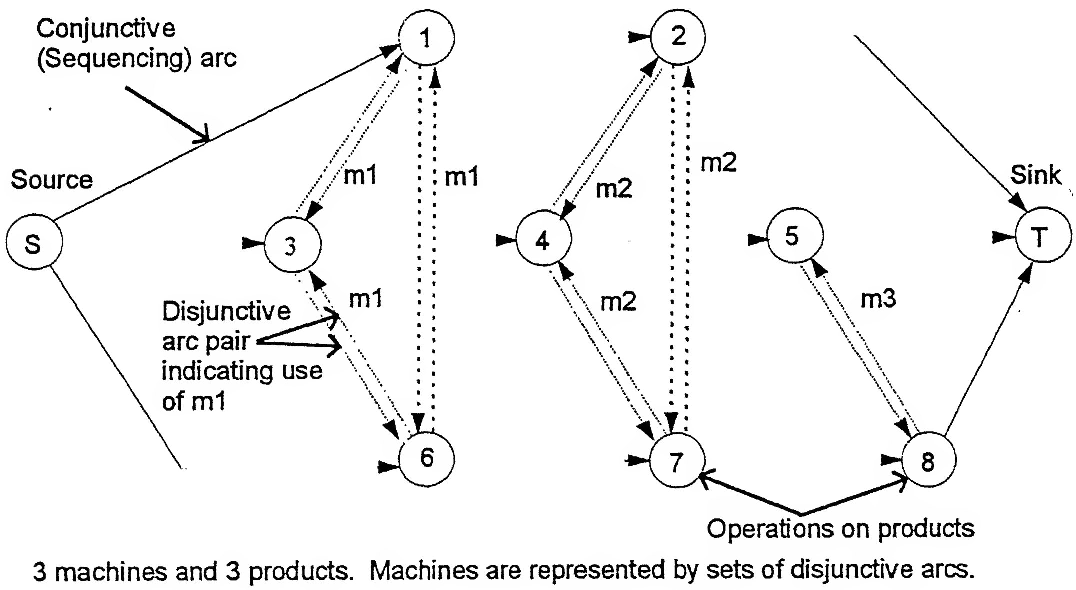


Figure 2-1 Network representation of the Standard Job Shop.

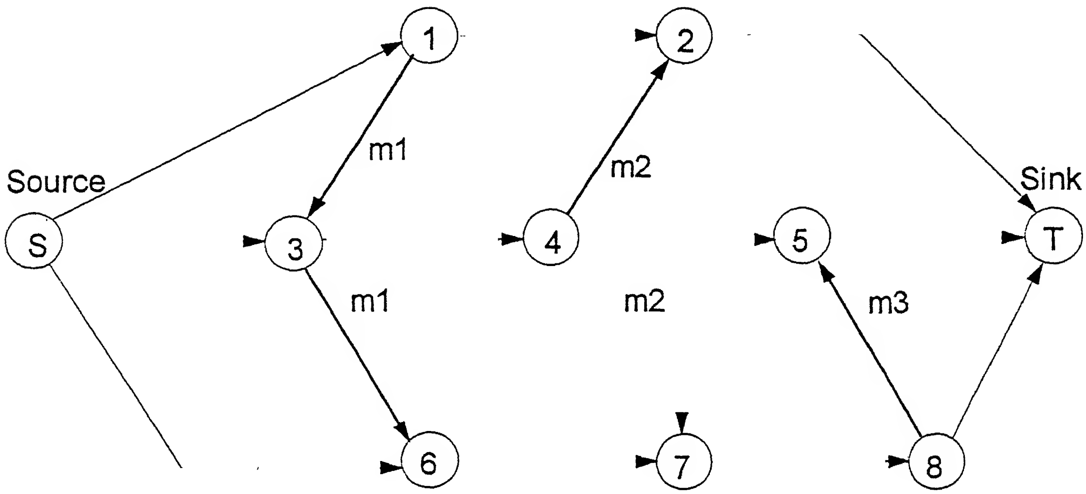
2.2.1 Representation of the problem

Balas represented the standard job shop problem in a graphical form using **nodes** and **arcs**. Each node except the first node and the last node represents an operation. The first and the last nodes are dummy operations — they just provide a starting point and termination point for the whole project.

Arcs are of two types. Conjunctive arcs (implying sequentially connecting) are well-defined and fixed arcs i.e., they cannot change their direction during the course of solution (see Figure 2-1 Network representation of the Standard Job Shop.). These arcs show the *precedence relationship* between those operations which are to be done on the same product in a technological sequence. Thus these arcs represent the sequence of processing required. Disjunctive arcs exist in pairs — both of them connecting the same nodes but pointing in opposite directions. To produce a feasible solution at most one of them is selected. Each disjunctive arc pair corresponds to a particular *resource*. Each resource is represented by a group of pairs of disjunctive arcs. For example, the three pairs of disjunctive arcs connecting operations 1, 3 and 6 in Figure 2-1 represent the resource $m1$. The single selected disjunctive arc (two from each pair) with its pointing arrow will indicate which operation (from the two ends nodes) will be processed first on that resource.

A feasible solution will be represented by the graph when *at most* one arc from each pair of disjunctive arcs is chosen in such a way that there are no cycles in the graph (Figure 2-2

Network representation of a feasible Standard Job Shop schedule.). Notice that each *pair* of disjunctive arcs in the figure has been replaced by a 'bold" arc, indicating the order in which the resource will be used. It is to be noted that the makespan of this schedule (feasible solution) will be the longest path (critical path) from the first node to the last node.



A feasible schedule after selecting atmost one arc from each pair of disjunctive arcs so that no cycles are formed in the graph.

Figure 2-2 Network representation of a feasible Standard Job Shop schedule.

Therefore, now the problem is finding which arc from each disjunctive pair which should be selected so that the length of the "critical path" in the resulting graph is minimum.

2.2.2 Solution

The algorithm proposed by E. Balas systematically enumerates all the possible selections, avoiding those about which it can be conclusively said that they will not result in reduced makespan. The algorithm is as follows:

Definitions:

Complementing: Reversing the direction of an arc i.e., selecting the alternate arc from a pair of disjunctive arcs in place of the existing arc. Only the members of disjunctive arcs can be complemented.

Δ: A quantity used to help decide which arc to select for complementing. Roughly speaking, it is the change in the length of the critical path if (only) that arc is complemented (the arc with which Δ is associated). Δ is calculated using a method given by Balas.

F: Set of fixed arcs.

Fixing: An act which *includes* an arc in the set F. After this, this arc cannot be complemented until made "unfixed" i.e., excluded from F.

$\boldsymbol{v}$*****: Least makespan found so far.

$\boldsymbol{D}$: the set of disjunctive arcs on the critical path.

Initial conditions:

$v^* = \infty$

All disjunctive arcs are "unfixed" in the beginning.

$F = \phi$ (the empty set)

Step 1. Start with any feasible solution. (Balas provides a method for finding a feasible solution.)

Step 2. If the length of the critical path in the graph with only fixed arcs and sequence arcs (connecting the operations bound with technological constraints) is greater than v^*, backtrack (Step 6).

Step 3. Calculate the length of the critical path (CPL) with all arcs (sequence, fixed and unfixed) included. If $CPL < v^*$ then $v^* = CPL$.

Calculate Δ for each disjunctive arc on the critical path that is not fixed i.e., $arc \notin F$, $arc \in D$. Locate the arc with minimum Δ: $arc_{min\,\Delta}$. If there exists no arc on the critical path such that $arc \in D$, $arc \notin F$, backtrack (Step 6).

Step 4. Complement $arc_{min\,\Delta}$. Include $arc_{min\,\Delta}$ in F:

$F = F \cup arc_{min\,\Delta}$

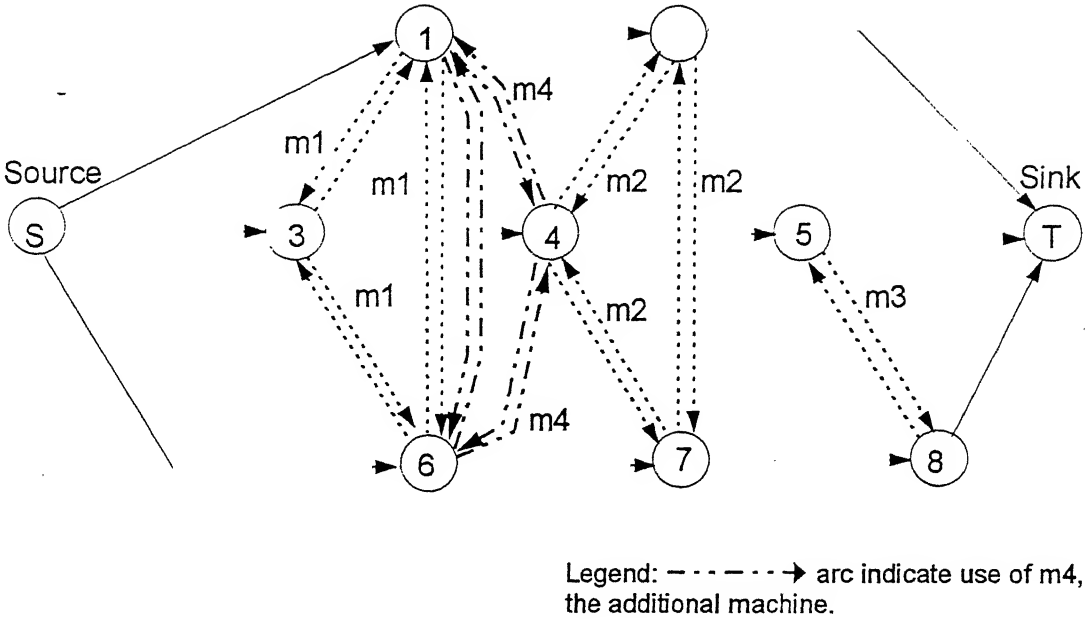
Step 5. Go to 2.

Step 6. Backtracking: Remove the last arc complemented from F. Again complement that arc and include the resulting arc in F.

If we are again back to the solution with which we started, and we have come here from Step 3, then the algorithm ends here.

2.3 An Extension of the Balas' Method to Consider Multiple Machine requirement in One Operation

While proposing the above algorithm, Balas assumed only the simple job shop scheduling problem constraints to be present, specifically, that only *one* machine is required in any operation. Therefore, all the disjunctive arcs coming out of or going into a node connect here only those nodes (operations) which are to use same machines (For instance, nodes 1, 3 and 6 in each use only $m1$.) However, a simple generalisation can be made to this representation. If there are other resources also e.g., jigs, tools, dies, operators etc. which are also used simultaneously on the same operation and if they are also required on other operations (thus the need for *their* scheduling as well), then each such requirement can be represented by a separate and distinct set of disjunctive arcs which interconnect all the nodes requiring that resource (Figure 2-3 Network representation of Job Shop problem with multiple machines in one operation.) Network representation of such a job shop problem shows multiple machines required in one operation. Operation 1 and 6 require both $m1$ and $m4$.



Operations 1 and 6 require m1 as well as m4. Similarly operation 4 requires m2 and m4.

Figure 2-3 Network representation of Job Shop problem with multiple machines in one operation.

The method for solving the extended (multi-machine) problem remains identical to Balas' except for one important change. In this enhanced and more general representation it is quite possible that there are some pairs of nodes which are connected together with *more than one* pair of disjunctive arcs, for example, operation 1 and 6 in Figure 2-3. Therefore, we need to redefine the term *complementing* as follows: Whenever the disjunctive arc connecting any two nodes *a* and *b* is to be complemented, complement *all* the disjunctive arcs connecting *a* and *b* together. Furthermore, while constructing the initial feasible solution, care must be taken so that of all the disjunctive pairs of arcs connecting operations *a* and *b*, the selection should be made in such a way that all the selected arcs are pointing in the *same direction*.

Proof:

The proof of the Balas' original method for the SGS is based on five prepositions provided by him [1]. Since the solution method in this enhanced case remains identical to that of Balas, it will suffice to show that all those five prepositions hold for the modified problem also. Of all the prepositions, only preposition number 2 needs to be changed. Balas' five original prepositions are as follows:

(i) In any network representation of the feasible solution (a feasible schedule), for any disjunctive arc (i, j), there is a path from source to sink containing (i, j).

(ii) Let C be the critical path in the network representation of any feasible solution and (i, j) is any one disjunctive arc on C. Another graph which will result from complementing (i, j) will also be circuit free, hence another feasible solution.

(iii) Let graphs G and G' represent two different feasible solutions with C and C' as their respective critical paths.

Let (i, j) be any disjunctive arc on C'. If $C < C'$ then G contains the complement (j, i) of at least one arc (i, j).

(iv) The value $\Delta\ (i, j)$ (see section 2.2.2, page 24) is well-defined, i.e., it is possible to calculate $\Delta\ (i, j)$, for all the disjunctive arcs in the feasible solution.

(v) Let (i, j) be an arc on the length of the critical path C of a feasible solution. Let C' be the length of critical path of another feasible solution obtained from the first feasible solution by complementing (i, j). Then

$$C' = C + \Delta\ (i,j) \qquad \text{if } \Delta\ (i,j) > 0$$

$$C \geq C' \geq C + \Delta\ (i,j) \qquad \text{if } \Delta\ (i,j) \leq 0$$

The role of these prepositions, their explanations and their proofs are given in [1]. The preposition (ii), after modification, may be stated as follows:

(ii) Let C be the critical path in the network representation of any feasible solution and (i, j) is any one *set* of disjunctive arcs which connect the nodes i and j on C. Another graph which will result from complementing the arcs (i, j) will also be circuit free.

The proof of the above preposition is also identical to that given by Balas for his preposition (ii). It may be briefly stated here: If complementing the set of arcs (i, j) results in a cycle in the graph, then it implies that there is another path from node i to node j which is not on C. The lengths of all the arcs emerging from i are equal to the processing time of the operation represented by the node i. Therefore the length of the

alternate path will be at least as much as (i, j). Since (i, j) are on the critical path, the nodes i and j are already at the maximum possible distance from each other. Hence by contradiction, there is no other path from node i to j than the one represented by the set (i, j) and therefore no cycle will be formed if the set of arcs (i, j) is complemented.

2.4 Solving Open Shop Scheduling problems

We have found that with this generalisation i.e., with the ability to solve the job shop problem with multiple types of resources in an operation, it is possible to model and solve a more general case of open shop scheduling problem. An open shop is described as follows.

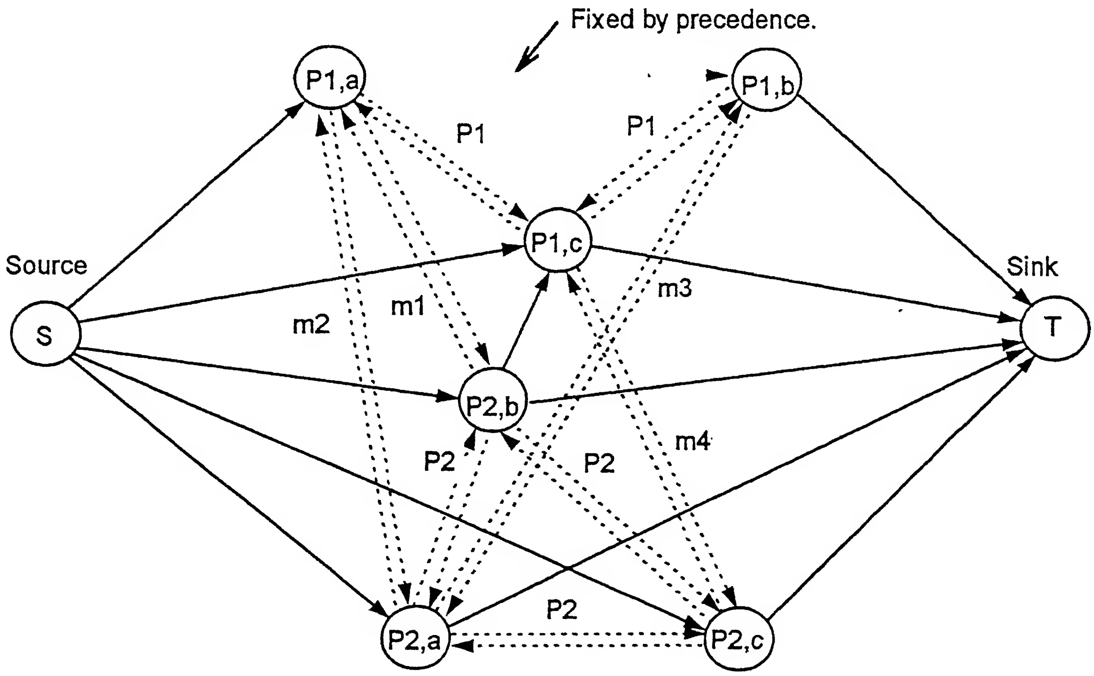
2.4.1 The standard open shop [3]

There are p products to be made using q different machines. Each product needs to be processed on some of the q machines. There is *no specific (technological) sequence* of processing required for any of the products. The problem is to determine the order of processing of operations on products on each of the q machines so that makespan is minimised.

The general problem of open shop scheduling which can be solved with the modified Balas method as follows.

There are p products to be made using q resources. Each product needs to processed on some of the resources and more than one type of resource may be used in each operation. The operations may have partial precedence relationship, i.e., some operation may be carried out in any order while others may have precedence relationship among them. The problem is that of determining the order of processing of the various operations fresh product on each of the q machines so that makespan is minimised.

2.4.2 The network representation of the open shop



Network representation of an Open Shop problem. Note the disjunctive arc linkages of operations on the same product. No precedence relationship exists between *P*1,*a* and *P*1,*c* or between *P*1,*c* and *P*1,*b*. However, *P*1,*a* preceds *P*1,*b*.

Figure 2-4 Network representation of a General Open Shop.

We first fix the source and the sink nodes as two dummy operations with zero processing times. Next, all operations having precedence relationships are linked with conjunctive arcs showing these relationships. All first operations are linked with the source through a conjunctive arc and similarly each last operation is linked with the sink. Those operations having no precedence relationship are linked directly with the source and the sink through conjunctive arcs.

Operations which are to be carried out on the *same* product are linked with each other through the pairs of disjunctive arcs except when the precedence is already established amongst them through conjunctive arcs. In the same manner, operations to be done on the same machine are linked to each other through pairs of disjunctive arcs (Figure 2-4 Network representation of a General Open Shop.).

For example, there are two products $P1$, $P2$ and they have operations as shown in the following table:

Table 2-1 Machine requirements of operations in a general open shop.

$P1$	operation a	operation b	operation c
m/cs	$m1$	$m3$	$m4$
used	$m2$		
$P2$	operation a	operation b	operation c
m/cs	$m3$	$m1$	$m4$
used	$m2$		

There are only 2 precedence constraints:

$P1, a \rightarrow P1, b;$

$P2, b \rightarrow P1, c$

Exactly the same method as described for the job shop scheduling with multiple types of resources in an operation, can be applied on this representation also to get an optimal solution.

3. A Genetic Algorithm for the Generalised Job Shop

In the last chapter we discussed Balas' network representation approach for solving the problem. A close look at the special features of the problem (Availability of multiple resources, each operation using one or more resources, one or more units of each type of resource, release dates, priority, partial ordering and non-availability of resources.) suggests that it would be difficult to extend the network representation approach to cover all the features of our problem. Balas attempted to do that in [4], but he presented only a formulation of the problem. He probably did not generalise his solution beyond the simple case as is apparent from his personal communication dated 22nd November 1995 on my enquiry of 1st November 1995, as follows:

> Dear Ankur Bhatnagar:
>
> The paper on Project Scheduling with Resource Constraints, written by me in the sixties, had the purpose of formulating the problem in the context of disjunctive graphs. The paper did not claim that the model is solvable by the same algorithm that solves the machine sequencing problem, it just said that it is amenable to the same type of implicit enumeration technique. The elaboration of such a technique was left to a

later paper, which - alas - was never written, since I ceased to work on this problem. But since you ask - and kind of hold me responsible for not following up on this matter - here is the way I see it.

The model is amenable to the same type of technique as the machine sequencing problem in this sense: in both cases, we have a disjunctive graph in which we are seeking a feasible selection among the disjunctive arcs, that minimizes the length of a longest path. The difference consists in what constitutes a feasible selection. In the case of machine sequencing, it means a subset of the disjunctive arcs containing just one member of every pair. In case of your problem it means a subset of the disjunctive arcs satisfying the resource constraints. In principle, you could generate every such subset, solve for each of them the longest path problem, and select the one for which the resulting path is the shortest. This of course would be inefficient. Implicit enumeration means that you develop various kinds of tests that allow you to make a large part of the enumeration implicit rather than explicit. Such tests have been developed for many problems; for this particular one, as far as I know, not much work has been done so far along these lines. Maybe you will try. Anyway, I wish you good luck and if you have further questions, don't hesitate to write. Regards, Egon Balas

Therefore seeking an alternative to implicit enumeration method would be well advised. Earlier we abandoned the Shifting Bottleneck Heuristic for similar reason. Such experience suggests that it is probably not possible to find a ready made solution method from

the literature and extend it to fulfill the needs of the generalised job shop. A search was carried out using OR abstracts and references of related papers written in the last six years on this subject for a suitable method to solve the GJS without success. It would therefore appear that some new method would have to be devised, perhaps from scratch, to tackle the unique characteristics of the generalised job shop scheduling problem. At this juncture an application of genetic algorithms was explored.

There was one more reason which encouraged us to seek a heuristic. The job shop scheduling problem in its simplest form is known to be strongly *NP* hard problem [21]. Since the method would be used to solve the scheduling problems by people who may not be willing to wait for too long for the single correct answer and also the obvious impact of such exact methods on costs of computing, looking for a *globally optimal solution* may not be always necessary. However, we would support the quest for optimal solutions even for GJS for numerous other reasons.

The GA seems to offers one practical advantage. At the shop floor sometimes there is no clear cut understanding as to what should be the precise form of the objective function that should be used for optimising. Of course, factories want the schedules with lower makespan which leads to better utilisation of resources but they usually also want the schedules with minimum weighted completion time which differentiate between a less important product from a more important product thus providing better service to the customers. In other words they might like to settle on the objective function itself by trial and

error. It is possible that management will like to develop a "feel" of the parameters of schedules generated using different objective functions over an extended period of time. It may be noted, GA, can be made to work regardless of the objective specified.

Thus, in contrast to "dedicated" analytical methods valid for particular objective functions, the GA provides certain flexibility in the use of objective function. We may simply substitute another objective function in the GA program and see the effects on the resulting schedules, without much trouble at the site.

However, there are some disadvantages with GA. Since genetic algorithm is based on randomisation, there is no repeatability in the method, unless all randomisation seeds are controlled. Therefore, when the program is run again for the same problem, it may come up with a different best schedule every time. This may result in some confusion among the people using the scheduling system. In order to handle this problem, the users must be adequately warned of this possibility in advance. Additionally, to operate effectively the GA must be optimally parameterised [37].

3.1 The Genetic Algorithm

As mentioned in section 2.1, the Genetic Algorithms (GA) are essentially the summary of nature's process of evolution of life on earth. Over a period of billions of years, earlier living organisms which were far simpler and worse in adapting to the environment have

developed into much more versatile, complex and robust organisms. This has happened due to natural selection as proposed by Darwin. Genetic Algorithms simulate this process of evolution to evolve better solutions of a problem from poor solutions. Here also the analogy of individual, population, offspring, natural selection etc. is used. The genetic algorithms were invented by Holland in 1975 [31]. For an interesting overview of the GA, reader is referred to [32].

The genetic algorithm used can be represented as follows:

1. Generate first generation (Initial population).
2. Evaluate the population and output the statistics.
3. Apply the crossover operator to the population.
4. Evaluate the population and output the statistics.
5. Evaluate terminating condition
6. If terminating condition is met, go to 10.
7. Apply selection operator to the population.
8. Apply mutation operator with a small probability.
9. Go to 3.
10. Sort the population; output the results.
11. End.

3.1.1 Initialising

The process of genetic algorithm starts with a *set* of initial solutions. Each solution in this set is called an individual in the GA terminology. In a combinatorial optimisation problem such as job shop scheduling, each individual is characterised by a particular permutation of feasible solution possibilities. The GA process starts with the set of individuals (the set henceforth referred to as population) created such that all possible feasible permutations corresponding to respective individuals may be generated randomly. Such initial population is called "first generation". All individuals are now evaluated for their "fitness". The fitness of an individual is the objective function value corresponding to that individual. The statistics describing the population, such as fitness of the best individual, average fitness of the population etc. are now computed.

3.1.2 Crossover

The next generation is produced from the population in the present generation by several "genetic" operations. One method uses *crossover*. Any two individual are selected from the population randomly and they are combined together using certain crossover *rules* to produce new solutions called daughter solutions which are (potential, depending upon *selection* — to be described later) members of the next generation. The two individuals (parents) are combined in a way such that the resulting solution will have some characteristics inherited from the first parent and some characteristic from the second parent. There

have been several rules that have been used by the researchers. A brief summary of such rules is presented here:

There are several types of crossovers which are being used in GA and new ones are being rapidly created [25]. A summary of some of them is given here. A comment on the feasibility of solutions resulting from each of the crossovers has been included. The feasibility depends upon the representation of the solutions as strings. The representation scheme itself is explained in section 3.2.1.

1. One point crossover

This crossover has been used and tested for its performance in this present scheduling project. Its method will be explained in detail later.

2. Two point crossover, Version 1

Two points are randomly selected on one of the parent strings dividing it into 3 parts. The first part and the third (last) part are taken as such and placed at the beginning and end of the child solution respectively. The middle part of the child is constructed from the remaining operations (not yet in the child) from the second parent while maintaining the order of their occurrence in the second parent.

Children in this crossover will always be feasible in terms of precedence relationships because, among other reasons, the operations in the first part of the parent 1, middle part of parent 1 and the last part of parent 1 remain in the same respective parts in the child.

3. Two point crossover, Version 2

Here also the parent is divided into 3 parts but instead of the first and the third part, the middle part is taken from it and placed at the same position in the child string. The first part and the last (third) part of the child string is constructed from the remaining operations from the second parent while maintaining the order of their occurrence in the second parent.

This method will not always yield feasible solutions in our case. This is so because here if there is any *precedence* relationship between an operation in the middle part of the parent 1 string and the operation in the third part of the parent 1 then there is no guarantee that the operation in the third part of the parent 1 will always occur in the third part of the child. Same is true with operations in the first part and in the middle part.

4. Two point crossover, Version 3

This is the mixture of the above two-point crossovers. These crossovers are applied to each pair of parents with some probability (for example, 0.5 for each version of crossover).

This method will also not yield feasible solutions because some of the children will be generated using the two-point crossover, version 2.

5. Two point crossover, Version 4

In this crossover operator, the child is divided in three parts based on two randomly chosen points. The first part of the child is constructed from the operations taken from the parent 1 while maintaining the sequence of parent 1. The second part is constructed from the operations in parent 2 selected from the beginning of parent 2 while maintaing its sequence. The third part is constructed from parent 1 by selecting the remaining operations from it while maintaining the order of their occurance in parent 1. This crossover operator is also used and tested in this project and will be explained in detail later.

6. Position based crossover [25]

Parent 1:	$\underline{a}$-b-c-$\underline{d}$-$\underline{e}$-f-$\underline{g}$-h
Child:	$\underline{a}$-c-b-$\underline{d}$-$\underline{e}$-f-$\underline{g}$-h
Parent 2:	c-b-f-e-a-h-g-d

The position of the randomly chosen operations in parent 1 is fixed in the child string also. The remaining operations are taken from parent 2 maintaining their order of occurrence in

parent 2. This method also may result in infeasible solutions in our case because of violation of precedence constraints.

Other methods which have been mainly used for solving TSP [23] are:

7. **Edge recombination crossover**
8. **Enhanced edge recombination crossover**
9. **Partially matched crossover**
10. **Cycle crossover**

3.1.3 Selection

At the end of crossover operation, we are left with a set of individuals comprising two groups of individuals: parents and daughter solutions. The population size has been increased in the process. In order to deal with a restricted set of individuals, so that the algorithm remains manageable, certain individuals are chosen and others dropped so that we retain the same number of individuals in the population that we had before crossover. The process of *choosing* the individuals is called *selection*. In GA selection is to be carried out using specific rules as suggested by Goldberg, inorder to condense optimisation. The rules may be:

1. Selecting the elite fraction — the best fitness individuals present in the expanded population produced after crossover.
2. Tournamenting — Repeating the process of choosing two individuals at random and then selecting the best individual among them.

3.1.4 Mutation

In GA, the next generation results from the previous through the process of crossover of parents. Crossover aims at preserving some of the characteristics of the parents into the daughter solutions, thus into the new generation. If GA is run as such, the *variety* of solutions (individuals) in populations of successive generations tends to diminish. This may result in restricting the search space and the GA process may prematurely converge to a local minima. Therefore it becomes necessary that we introduce certain changes in the individuals through a process other than crossover (not depending upon parents) to avoid such possibility. This process of introducing random and additional changes in the solution charaterstics is called *mutation*.

The logic of producing new individuals through crossover is based on the expectation that the combination of two good solutions is *likely* to result in a better solution. In order to preserve this logic, mutation ought to be a rare event.

Predominantly, a few particular types of mutation operators have been used extensively in the works involving genetic algorithm so far. While choosing the mutation operator for scheduling problems, following four types of the mutations have been considered [25]:

1. Adjacent two job change
2. Arbitrary two job change
3. Arbitrary three job change
4. Shift change

In Adjacent two job change, any random job is chosen and interchanged with next adjacent operation in the sequence.

Any two operations in the sequence, not necessarily adjacent to each other, are selected randomly and interchanged with each other in the arbitrary two job change.

In arbitrary three job change any three operations in the sequence are selected and interchanged among each other in a random order. The above two mutations are special cases of this mutation operator.

Shift change: An operation is selected randomly from the sequence. Another position is determined randomly in the sequence and then the selected operation is deleted from the former position and inserted at the latter position.

When we consider the precedence constraints that exist in the GJS problem, we find that we cannot implement *any* of the above mutation operators as such. The reason for this is that whenever we randomly change the position of an operation relative to another in the sequence, the sequence of assignment of these operations on the *resources* common to both of these operations will also change. And, if these two jobs happen to be related to each other through precedence constraints, that will result in an infeasible sequence or schedule.

Therefore, in order to implement any of these operators, we must also have a mechanism which quickly checks whether the resulting sequence is a feasible sequence or not.

In the particular case of *adjacent two-job change*, this check can be made by examining only the immediate successor of the first job. Since it is relatively easy to make such check in terms of the required computation effort in the adjacent two jobs change mutation case, this particular mutation has been implemented in the present work. In any of the above mutation methods when any of the operation is moved from one place in the string to another, the computation would involve checking the precedence relationship of the operation with each of the operations relative to which it is moved and if any of them is

violated, that particular mutation attempt would have to be abandoned. Therefore, the mutations involving a large degree of job shuffling would anyway become more rare to occur.

3.2 Representing the GJS "solution" (schedule) in a form adaptable to the GA methodology

Before we look at the representation of the problem, we must recall the forms of *solutions* which are amenable to genetic algorithm. One desirable feature of such a form is that it should be possible to "crossover" any two feasible solutions to produce another feasible solution (daughter solution). However, the GA can be used even if the daughter solution is not feasible by placing a penalty on such solutions, though one would rather prefer to have *feasible* daughter solutions. Resulting in feasible daughter solutions will depend upon the exact form of representation and the method used for crossover. It is also desirable that the crossover method itself be not too complex. This is so because one objective in crossover is to preserve a part of the *essence* of the goodness of both the parents and pass it to the daughter solution.

Another requirement on the form (though not as restrictive as the one above) is that the form should be amenable to mutation i.e., a simple arbitrary change in the solution should still produce a feasible solution.

Still-another requirement is that the form should be subject to randomisation. This means that it should be possible (at least theoretically) to generate *all* the feasible solutions randomly and with equal probability.

In most of the implementations of the genetic algorithm the form used to represent scheduling-type solutions is sequential. A sequence of variables representing objects or 0s and 1s is often used for example, to represent the solutions of flow shop scheduling problems [22], traveling salesman problems [23], knapsack problems [24] etc. These representations fulfill the above stated requirements. Thus if one could represent the solutions of the GJS problem also in the form of some such sequence that meets the above requirements then one might attempt to use the genetic algorithm to solve the GJS. The network representation (Figure 2-3) does not seem to hold such possibility.

3.2.1 A representation of the GJS problem

The issue of "representation" is akin to coding the problem as shown by Goldberg. To start with, one may visualise a feasible solution to the generalised job shop problem including all its special features (Chapter 1) in the form of the familiar Gantt chart. There is a collection of resources in the GJS and in front of each resource, there is a queue of operations assigned to it. Altering the order of the operations in these queues will result in different schedules. This Gantt chart type representation augmented with queues can be used

for cases when the resources are not available during certain intervals, operations involving different types of resources in different numbers, availability of more than one resources of the same kind, precedence constraints, etc. For example, if a particular resource is not available from time t_1 to t_2 then one may mark the time slot in front of that resource from t_1 to t_2 as "*Down*" or "*Not Available*". However, the queues in front of each resource are not easily amenable to genetic operations as it would take a large number of queues to represent just one solution and additionally and importantly, the operations in these queues are often interrelated.

Visualise the shop again. At a given point in time there is a set of operations which we are trying to schedule. There is another set of operations which we are *not* trying to schedule; these were already scheduled on the resources earlier. These operations might not be considered for scheduling at a particular point in time because of two reasons: (1) the operations have already begun and hence they cannot be interrupted now; and (2) because of certain extraneous reasons, management wants to process these operations at predecided times (may be the customer is coming to see the processing at a specific time). We assume further that all operations are *re*schedulable except those that have been completed and those started in the past and are still continuing. Therefore we may reschedule all those operations which haven't started yet unless specifically marked for exclusion by management. We also assume that the periods of nonavailability of a resource are represented by special "operations" which will start on the resource when the period of nonavailability starts and their "processing times" will be the same as expected downtimes.

Such "operations" are also not reschedulable. We use these ideas to develop an enhanced Gantt chart type representation of a solution as below.

3.3 Solving the Problem

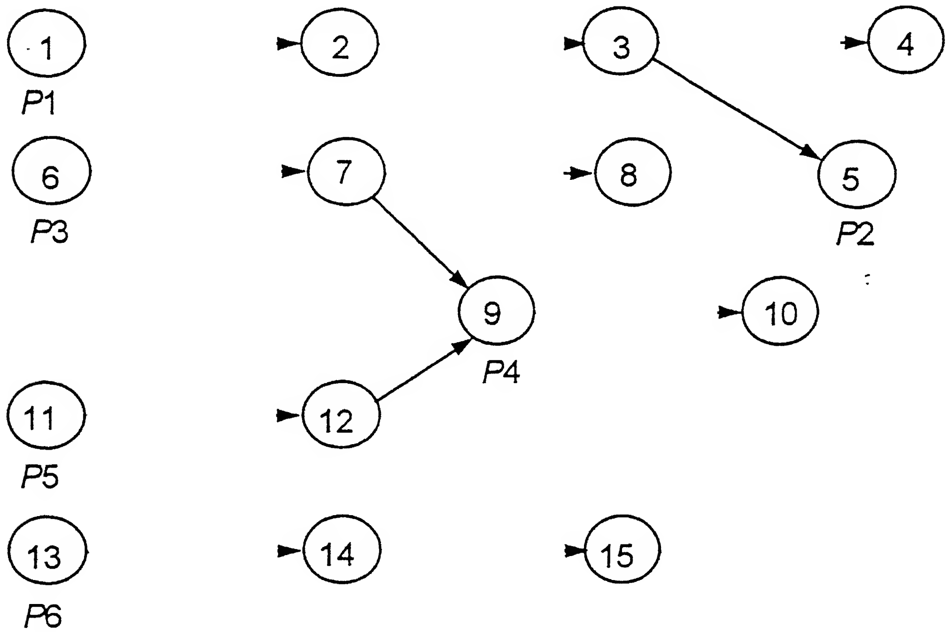
We first define a "special" schedule (in the form of a Gantt chart) which contains only those operations which were started in the past (not reschedulable), the planned down-times (not reschedulable), and those operations which were specifically marked as not re-schedulable (For such operations all their predecessors too will be included in the special schedule, marked not reschedulable). We call this schedule as *Resource schedule* because this schedule carries the list of all the associated resources along with the duties (not to be scheduled) assigned to each of them.

We begin with a *copy* of the resource schedule to proceed further. In the next step we generate a random, feasible GJS schedule of the schedulable operations as follows.

First, we generate a sequence of operations called the *Assignment Sequence*. The assign-ment sequence is used for producing the GJS schedule in the form of a Gantt chart.

3.3.1 Creating an assignment sequence

The method for creating the assignment sequence may be illustrated using the set of operations shown in Figure 3-1. The nodes and arcs show the operations and their precedence relationships respectively. Precedence relationships in the GJS may be fixed either by technological requirements or by product structure in the MRP sense. There are six products marked $P1$, $P2$, $P3$, $P4$, $P5$, $P6$ to be made here. Let O be the set of all operations to be scheduled. $O = \{1, 2, 3, ..., 15\}$. Let L be those operations which don't have any of



$O = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
$L = \{1, 8, 12, 14\}$
A = 11–6–13–7

Figure 3-1 A particular stage during the construction of an assignment sequence. An operation may be now chosen from L and placed at the end of A.

their *predecessors* in the set O, $L \subset O$. Initially $L = \{1, 6, 11, 13\}$. These are all schedulable. Choose any one of these *schedulable* operations randomly from L. Let that operation be operation number 11. The chosen operation is placed in the assignment sequence A (initially A is empty) and removed from L. Next, we place all the successors of the chosen operation (11) in L (except those whose other predecessors do not exist in A and whose predecessors have not already been scheduled by the management — thus not schedulable) and place them in L. Thus $L = \{1, 6, 12, 13\}$ and $A = 11$.

We again choose an operation from the list L randomly and place it at the end of the existing assignment sequence A. Let this operation be operation number 6. Therefore A becomes 11-6 and L becomes $\{1, 7, 12, 13\}$. We keep on repeating this procedure (See Figure 3-1.) until there is no operation left in L. At the end of this process we would produce a complete assignment sequence in A.

The feasible schedule is finally generated at the end of the following process which assigns the members in the assignment sequence to the required resources.

After completing the construction of the assignment sequence we assign the operations in the assignment sequence to the resource schedule while taking care to preserve the *order* in the assignment sequence. This produces a feasible solution. We carry out assignment of operations to resources in such a way so that we get an "active schedule". An active schedule is a schedule in which an operation cannot be expedited without delaying some

other operation. The method of assignment is described below. The following terms are used in this description:

Resource type: An attribute of an operation indicating the type of resources it requires. If an operation requires machinists, lathe and clamps, then "resource types" for this operation will be machinists, lathes and clamps.

Resource identification: Each resource in the shop has its own "identifier". If three lathes are available, these may be identified as lathe *a*, lathe *b*, lathe *c* and then *a*, *b*, *c* become their respective identification or tags.

Number of resource type required in an operation: An operation may require three operators and two clamps. Then 3 and 2 are the respective number of resource types required in that operation.

3.3.2 Method of assignment of the operations to resources to form active schedules

This method simulates the dynamics of the GJS. It has been illustrated in Figure 3-3.

1. Select the next operation (starting from the head of the assignment sequence) from the assignment sequence. Say, this is operation *o*.

2. If o is the starting operation on the associated product then the Earliest Start Time (*EST*) of o is set coincident with the product's release date. Otherwise *EST* is set equal to the finishing time of the latest of its predecessor operations.

3. Select the (next) type of resource which this operation requires (indicated in R_{type} of o); the number of such resources required by o is N_{res}. Set $N_{slots} = 0$. ('Slots" are defined later.)

4. Scan the schedule developed so far at the time *EST* for all the resources of type R_{type}.

5. For a resource of type R_{type} and identification R_{id}, check if a time slot exists starting at *EST* which can accommodate o. A slot is said to exist if the operation o can be started

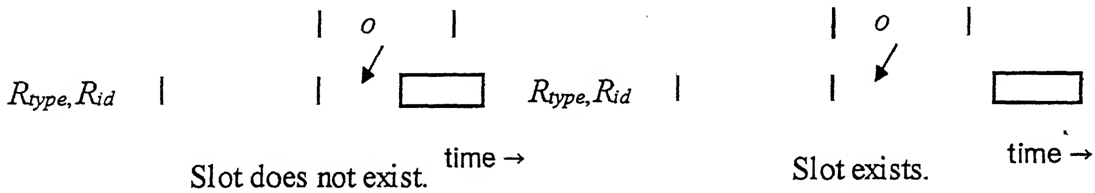


Figure 3-2 Illustration of a time slot on a resource for an operation.

at time *EST* on the resource R_{id} without delaying any other operation. Figure 3-2 illustrates instances where slots do not/do exist.

If a slot exists, increment N_{slots} by 1:

$N_{slots} = N_{slots} + 1$

6. Return to 4. until whole of the schedule is scanned.

7. If $N_{slots} < N_{res}$ then increase EST to the lowest finish time of operations *after EST* on all the resources of type R_{type}.

8. Go to 4. until $N_{slots} < N_{res}$.

9. If R_{type} is the last resource type in the list of resources required in o then

   R_{type} = First resource type in the list of resources required by o;

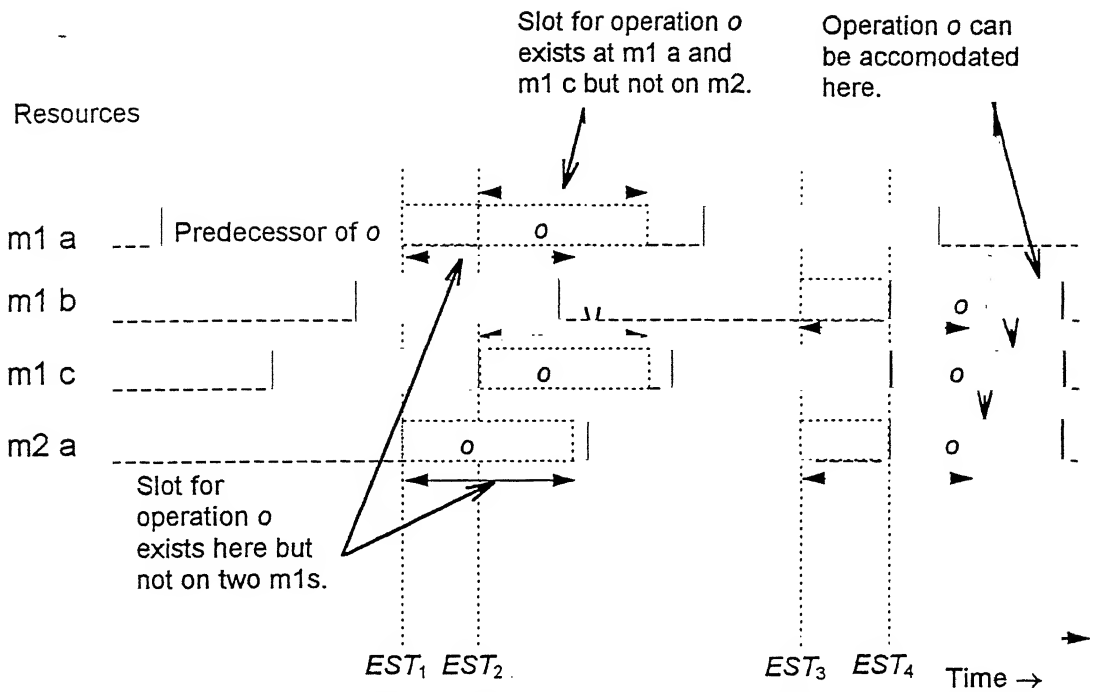
   otherwise R_{type} = Next resource type in the list.

10. Go to 3 until o is accommodated on all the resources of types required in o.

11. Assign o on all the resources of types (R_{type}s) required in o in their respective numbers (N_{res}s) at the time EST.

    If for any resource type the number of slots available at EST is more than the number of this resource type required, then assign o randomly among the available slots.

12. Return to step 1 until all the operations in the assignment sequence have been assigned.



This figure shows how *EST* changes when *o* is assigned to the resources required by it according to the method given in Section 3.3.2. *o* requires two *m*1 and one *m*2. Other operations are either pre-scheduled in the resource schedule or the operations assigned before *o*.

Figure 3-3 Illustration of method of assignment of the operations to resources to form active schedules.

3.3.3 Generating first generation of the population

For each individual of the initial population, the random assignment sequence is created using the method described above and then it is assigned to a copy of resource schedule (defined in Sec. 3.3). This way we get one individual (i.e., one feasible solution to the GJS) of the first generation.

Similarly, more random assignment sequences are produced and each of them is assigned to a fresh copy of the resource schedule to get a whole population consisting the first generation.

3.3.4 Evaluating the population

Each individual of the population is evaluated using the objective function (makespan or weighted completion time) specified. The objective function routine would scan the schedule and compute the objective function value of that schedule.

For each generation average value and the best value of the objective function is also computed.

3.3.5 Crossover

For crossover two individuals are randomly picked up from the population and are combined in certain ways to produce daughter solutions. We have examined the effect of two different types of crossover methods on the effectiveness of the program. They are one point crossover and two point crossover. Given two individuals A and B we can produce 4 daughter solutions using the same method in different ways. Here it is shown how two daughter solutions are produced for $A \times B$ using one point crossover.

3.3.5.1 *One point crossover*

One point crossover is carried out by performing following three steps:

1. Choose a random number between 1 and $N_{operations}$ where $N_{operations}$ is the number of operations in each of the two assignment sequences of A and B. Let that random number be r.

2. Choose first r operations from A's assignment sequence and place them in new sequence S as such.

3. Scan B's assignment sequence from the start. If an operation (in B's assignment sequence) is not there in S, add it at the end of S. Repeat this until whole of B's assignment sequence is scanned.

For example, let *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k* represent the operations. Let A's assignment sequence be: *a-b-c-d-e-f-g-h-i-j-k*. Let B's assignment sequence be:

c-b-f-g-a-j-e-d-i-k-h

Crossover sequence S for $r = 5$:

a-b-c-d-e-f-g-j-i-k-h

A B

The second daughter solution will be generated in the same way except that instead of starting from the *front end* of both the sequences as we did earlier, now we will start from the *back end* of both the sequences.

The front end of the sequence is defined as the left-most end of the sequence and the back end of the sequence is defined as the right-most end of it.

Therefore first r operations from the back end of B are selected as such and the remaining operations are selected from A's sequence also starting from the back end.

Reverse crossover sequence for $r = 5$:

```
a-b-c-f-g-j-e-d-i-k-h
     A           B
```

We will get two more offspring from A and B when we cross $B \times A$ in the same fashion as in $A \times B$ demonstrated just now. It will look like this:

Crossover sequence for $r = 5$:

```
c-b-f-g-a-d-e-h-i-j-k
     B          A
```

Reverse crossover sequence for $r = 5$:

```
c-b-f-a-e-d-g-h-i-j-k
     B           A
```

The above crossover operations retain feasibilty.

3.3.5.2 Two point crossover

This is similar to the one point crossover except that we start by generating two random numbers, r_1 and r_2, $r_1 < r_2$.

We take operations up to r_1 operations from the first sequence, up to r_2 operations from the second sequence and remaining again from the first sequence in the manner indicated in 1-point crossover. Here also we can obtain four offsprings, by permuting between A and B and the front end and the reverse end of the sequences.

For example, the crossover sequence for $A \times B$ with $r_1 = 3$, $r_2 = 8$ would be as follows:

```
a-b-c-f-g-j-e-d-h-i-k
  A        B       A
```

The reverse crossover sequence for $A \times B$, ($r_1 = 3$, $r_2 = 8$) will be:

```
b-c-f-g-a-e-d-h-i-j-k
  A        B      A
```

The crossover sequence for $B \times A$, ($r_1 = 3$, $r_2 = 8$) is:

```
c-b-f-a-d-e-g-h-j-i-k
  B        A       B
```

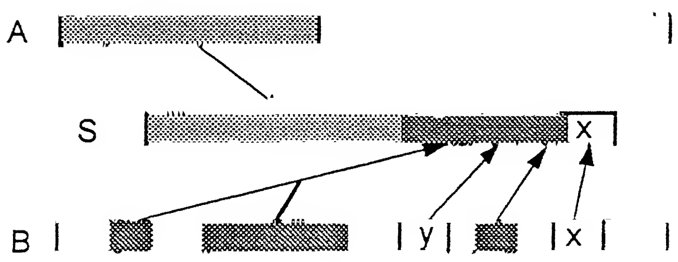
and the reverse crossover sequence for $B \times A$, ($r_1 = 3$, $r_2 = 8$) is:

```
c-b-a-d-e-f-g-j-i-k-h
  B        A        B
```

Once we have found a new sequence, the process of *assigning* the sequence is carried out to obtain a new individual.

It must be pointed out here that all the new sequences which are the outcome of cross-overs (both type) will result in feasible schedules, if the parent sequences result in feasible schedules. A proof is given below in support of this for the case of one point crossover of the first kind. The same proof applies to other kinds also.

Let A and B represent two feasible sequences. We take first r operations from A and place them in S as such (maintaining their sequence). Since A is feasible, S also will be feasible till first r operations. Now we start assigning operations from B from its front end to S. Let there be an operation x in B which does not exist in S. Therefore x would be placed at the end of S, according to the crossover method. If x has any predecessor, let it be y, then



Now placing x from B in S. Note that y will always occur prior to x in B. There are only 2 possibilties: y has come into S from A, and it hasn't. If it hasn't, it must come from B, prior to x.

Figure 3-4 Illustration of the feasibility of the daughter solutions if parents are feasible.

it has already come in S either from A or if not from A then from B itself. We can say so because since B is a feasible sequence, y will always occur prior to x in B. Since we are placing the operations from B into S from the beginning of B, therefore when we are placing x in S, we have already placed y in it. Hence in S also y will always occur before x. This means that predecessors will always occur before the successors in the new sequence hence the new sequence is also feasible.

3.3.6 Selection

In selection, for a certain percentage of population, the top elite individuals are selected from the previous generation. From the rest of the population, tournamenting is applied. In tournamenting any two individuals are (randomly) picked up from the previous generation and the better individual is retained. The other individual is returned back into the pool of unselected individuals in the population. It may again get picked up later.

3.3.7 Mutation

The adjacent-two-job change mutation operator has been implemented in this project. An operation here is chosen randomly from the given sequence of operations, excluding the last operation. All the immediate successors of this operation are examined to see if the next operation in the assignment sequence is an immediate successor or not. If the next

operation is not an immediate successor of the selected operation, then the two operations are interchanged.

It may be stated here that the resulting mutation will always be a feasible solution. This is so because if the next operation is not an immediate successor of the previous operation, then the two will not have any precedence relationship.

Say the operation chosen randomly is a. a is having b as the next operation in the sequence. If b is not an immediate successor of a then we may interchange a and b resulting in another feasible solution. To prove this assume that if b is not an immediate successor of a then let it be an indirect successor. That means that there must be at least one operation, say c which is a successor of a and a predecessor of b. Since the given sequence was feasible, therefore c must occur after a and before b in the sequence. And since there is no operation between a and b, therefore it is proved that such a situation is not possible i.e., if b is not an immediate successor of a then it will not be an indirect successor either and thus not related to a through any precedence constraints. And if a and b are not related through any precedence constraints, they can be freely interchanged to produce another feasible solution.

This genetic algorithm was coded to give it the shape of a program in the C programming language [36]. The program is portable and it can run under UNIX as well as DOS environment, provided a C compiler is available.

4. Parametric Optimisation of the GA

4.1 Objective

It is now widely reported [37] that the performance of the GA depends on its parameters. For any general GA these parameters are the population size, probability of crossover, type of crossover operator applied to the population, methods of selection or the mix of methods of selection, probability of mutation, types of mutation and so on. Another aspect of any process is the effect of noise on its performance. Noise is any factor which may have an influence on the process and is beyond the control of the user of the process. In case of GAs or any other optimisation method, the noise can be the different problems which same algorithm seeks to solve. It is beyond our control to enforce any kind of uniformity on the problems which are given to the algorithm to solve yet we would like our algorithm to perform at its best on as wide a range of problems as possible. We would prefer to make the implementation of the GA *robust* through *parameter selection* to minimse tuning the algorithm each time it is applied on a new problem. Experiments will have to be carried out on the GA for achieving robustness according to a predesigned scheme. Krishan Raman [37] was the first to apply Design of Experiments to parameterise the GA.

The objective of the parameterisation experimentation can be stated as follows:

1. To determine the parameters (or factors) which have any effect on the convergence properties of the genetic algorithm presented in this thesis project.

2. To determine those parameters whose interaction among themselves has any effect on the performance of the GA. This will help in selecting the parameters at such levels so that their effects do not counter each other.

3. To determine whether changing the problem has any effect on the efficacy of the algorithm.

4. To determine which factors have a dissimilar effect on the performance when the problem is changed i.e., the interaction of the factors with the change in the problem.

After finding the relevant factors, the values of each of them is to be decided.

The factors which we selected for examination, as suggested by the work of earlier researchers attempting to parameterise, are as follows:

1. Population size
2. Probability of crossover
3. Type of crossover

4. Probability of mutation

5. Percentage of elite individuals of the population retained in each selection

Each of these factors are tested for two objective functions namely, makespan and weighted completion time.

The robustness of the "optimally" parameterised GA was tested by solving two different problems for each combination of levels of the factors.

4.2 Design of Experiments

It must be admitted here that our objective to experimentally seek good parameter values does not go into finding the exact cause-effect relationships between the GA factors and the GA's response. The approach of Design of Experiments has been used only as an illustration of how one may attempt parameterisation. All the factors have been examined at only two levels and a partial factorial design has been used. This would not allow investigation into non-linear influence of parameters or the effect of three or more factor interaction. Therefore, at no stage in the experimentation and analysis of results we imply that the truly optimum combination of the levels of factors is being searched. Also, no intrapolation or extrapolation has been attempted. For getting a better understanding of effects of each parameters on the algorithm, experiments with factors at more than two levels and a full factorial design would be recommended, as done by other investigators [37].

4.2.1 Data Values

When the experiment was run for a particular problem for a particular objective function, the value of the objective was the raw output data. This raw data was transformed into the percentage excess of the best value noticed. For example, if while doing the experiments on problem *A*, for optimising makespan, the best value we got was 54 and the value of a particular run turned out to be 56 then the output 56 was transformed as follows:

$$\text{Transformed value of output (56)} = \frac{56}{54} \times 100 - 100$$

$$= 3.70$$

This transformation was considered necessary though it is highly arbitrary and not indicative of the GA's convergence performance, or suitable for quantitavely comparing the performance of the algorithm on two different problems.

4.2.2 The design of the experiment

The experiment was conducted in the standard DOE format in 'trials'. In each trial the factors were set at different levels as dictated by the experimental design. Each of the level was either level 1 or level 2. Therefore each trial can be represented in symbols by a sequence of 1s and 2s. For each trial, 5 replications were carried out for each problem and for each of the two objective functions: makespan and weighted completion time.

The designing of experiment aims at choosing the levels of each factor in each of the trial and analysing the data. In the following pages only the procedure of the experiment will be overviewed without explaining the theory [34] [26] underlying the design.

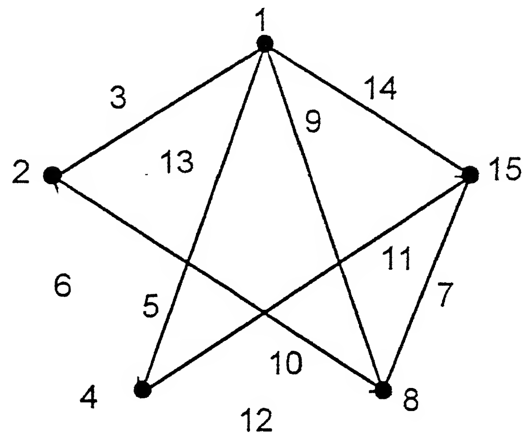
One prerequisite in many partial factorial designs is the *orthogonality*. Orthogonality usually allows the quick evaluation of the effect of certain factors independent of the influence of other factors or interactions. Othogonal Arrays are used for achieving orthogonality and for minimising the number of trials while still obtaining most of the required information. Of them the L16 Orthogonal Array (OA) was selected for the present experimentation (see Table 4-2). Each row in the array shows the settings for a trial and each column in the array may be assigned to a factor, depending upon the object of the experiment. The numbers in the rows show the levels of each of the factor assigned to a column in a particular trial. "Orthogonality" implies that for both the levels of any factor, all the other factors will have a setting which is same in terms of their number of levels, and thus the trials are balanced. This allows the easy estimation of certain factors assigned to certain particular columns independently. For example, in Table 4-2 column 1 is set at level 1 for first eight trials and at level 2 for next eight trials. For column 1 at level 1, any other column, let it be column 2, is set at level 1 four times and four times at level 2. For column 1 at level 2 also, column 2 is set at level 1 four times and four times at level 2. Therefore, effect of column 1 factor can be estimated independent of column 2. Since there are only 5 factors to be evaluated but 15 columns in the array, only 5 of the columns will be assigned to the factors. Depending upon the assignment of the columns for factors, other columns may represent

the effects of interaction among the factors. There are some linear graphs associated with each OA which show which column represents the interaction of which two columns. In this case, the linear graph in Figure 4-1 was used to assign the five factors to the appropriate columns.

Table 4-2 L16 Array ([26], page 215)

Col. →	i	ii	iii	iv	v	vi	vii	viii	ix	x	xi	xii	xiii	xiv	xv
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
3	1	1	1	2	2	2	2	1	1	1	1	2	2	2	2
4	1	1	1	2	2	2	2	2	2	2	2	1	1	1	1
5	1	2	2	1	1	2	2	1	1	2	2	1	1	2	2
6	1	2	2	1	1	2	2	2	2	1	1	2	2	1	1
7	1	2	2	2	2	1	1	1	1	2	2	2	2	1	1
8	1	2	2	2	2	1	1	2	2	1	1	1	1	2	2
9	2	1	2	1	2	1	2	1	2	1	2	1	2	1	2
10	2	1	2	1	2	1	2	2	1	2	1	2	1	2	1
11	2	1	2	2	1	2	1	1	2	1	2	2	1	2	1
12	2	1	2	2	1	2	1	2	1	2	1	1	2	1	2
13	2	2	1	1	2	2	1	1	2	2	1	1	2	2	1
14	2	2	1	1	2	2	1	2	1	1	2	2	1	1	2
15	2	2	1	2	1	1	2	1	2	2	1	2	1	1	2
16	2	2	1	2	1	1	2	2	1	1	2	1	2	2	1

The linear graph below shows that one may evaluate the interaction of all the main factors if we assign them to column numbers (1, 2, 4, 8 and 15) shown at the vertices.



A Linear Graph for L16 OA. There are several other such graphs which also exist for L16 OA [26], page 216.

Figure 4-1 A Linear graph for L16.

The numbers on the arcs show the columns which represent the interaction of the columns at the vertices at the ends of each arc. For example, column number 3 represents the interaction of columns 1 and 2. The same columns may also represent the effect of three-factor or higher levle interactions also. Therefore if there is a significant interaction among any three factors or more then it can confound the effects of other two-factor interactions or the main factor effects. Therefore we make an assumption that there exists no three-factor or higher level interaction. The assignment factors to the columns was done as shown in the following table:

Table 4-3 Assignment of factors to the columns of L16 OA.

Factor	**Column**
Population Size	1
Probability of crossover	2
Type of crossover	4
Probability of mutation	8
Percentage elite of the population selected	15

The experiments were conducted by varying the factors strictly according to the L16 OA. The experimental data was obtained in four parts as the experiments were conducted for two scheduling problems and for two scheduling objectives for each problem. Each GA "run" of each trial was run for 40 generations assuming that GA would then get sufficiently close the true though unknown optimum solution. The data recorded for each run was, respectively: the worst, the average, and the best value of the objective function for each generation; makespan and weighted completion time of the best individual; and the CPU time used in each run.

4.3 Analysis of experimental data

Raw data was analysed as shown in following stages, separately for the two objective functions. The analysis was done using *the best* value achieved of each objective function in the last (40th) GA generation.

Step 1.

In this step the best value of the objective function in the 40th generation in each of the runs (each DOE trial, each replication) were separated from the rest of the data. After this, these values were transformed as shown in Section 4.2.1 and laid out in a tabular form (Table 6-1 and Table 6-2). The rows correspond to the trials and the columns the replications. Two such tables were formed for the two problems shown juxtaposed

in Table 6-3. In Table 6-3, the second to sixth columns correspond to the five replications in the problem A for the respective trial. The last five columns correspond to problem B for the trials.

Step 2.

In Table 6-3 and Table 6-4, the five columns corresponding to the problem A are summed together and five columns corresponding to the problem B are also summed together. The two resulting tables (Table 6-5 and Table 6-6) each now have three columns. The first column in each table here indicates trial number, the second the sum of problem A transformed makespan or weighted completion time data and the third the sum of problem B objective function data. The rows correspond to the different DOE trials.

Step 3.

The columns corresponding to problem A and the columns corresponding to problem B in Table 6-5 and Table 6-6 are summed together to produce the Step 3 table (Table 6-7 and Table 6-8). The variation in the transformed data values is apparently only due to change in the factor levels across the trials and random errors, ignoring that even in 40 generations the GA might not have found near optimal solutions.

Step 4: Analysis of Variance (ANOVA)

Analysis of Variance is used to compare the means of populations of two normal distributions, assuming that variance is same in both of them. In the present application it is assumed that data corresponding to two levels of population (on the basis of any of the experimental factors (for example, populations corresponding to two levels of crossover types) or interactions) represents two nearly normally distributed populations.

ANOVA as we have used may be summarised as follows. [34]

Subscripts:

k = "problem" (we have two scheduling problems to be optimised using GA)

i = factor or interaction source of variability.

r = Experiment (DOE trial) number.

μ = The overall response or mean of the population including all trials, both problems and replications *for a particular objective function.*

σ_e^2 = Error variances; Error variance is the variance in the observation data due to random factors excluding the effects of any of the experimental factors including change of problem and interactions. In the present case seed of random number generator and effects of higher order interactions not included in the design are the sources of error.

T = Total of all the observations in the experiment.

$\overline{T}$ = Average of all the observations in the experiment.

= Least squares estimate of μ [34]

$\mu_{ij(r)}$ = Mean of population corresponding to factor i at a level j (j = 1, 2) in the rth trial.

$\tau_{ij(r)}$ = Deviation of $\mu_{ij(r)}$ from μ i.e., the effect of change in factor or interaction i at level j in the rth trial.

$= \mu_{ij(r)} - \mu$

x_{rkl} = A sample data point in the observations in the rth trial, kth problem, lth replication.

$\mu_{m_{ij(r)k}}$ = Mean of population corresponding to $m_{ij(r)k}$

$$m_{ij(r)k} = \begin{cases} 1 \text{ if } j(r) = k \\ 2 \text{ otherwise.} \end{cases}$$

for a two level experiment where i is factor/interaction, $j(r)$ is level of ith factor or interaction in experiment r and k is "problem".

Thus $m_{ij(r)k}$ represents the level of interaction of factor/interaction i with the "problem".

$\tau_{m_{ij(r)k}}$ = Deviation of $\mu_{m_{ij(r)k}}$ from μ i.e., the effect of interaction of factor/interaction i with the "problem", at level $m_{ij(r)k}$

$= \mu_{m_{ij(r)k}} - \mu$

μ_k = Mean of population corresponding to problem k.

τ_k = Deviation of μ_k from μ.

$\tau_k = \mu_k - \mu$ i.e., the effect of change in problem at level k.

e_{rkl} = Error of observation in x_{rkl} due to other factors and higher order interactions.

Any particular observation x_{rkl} is assumed here to be representable as the *addition* of population mean, deviations due to change in levels of all factors, interactions, problem change, and error:

$$x_{rkl} = \mu + \sum_{i=1}^{15} \tau_{ij(r)} + \tau_k + \sum_{i=1}^{15} \tau_{m_{ij(r)k}} + e_{rkl} \quad \text{— (1)}$$

Note that $\tau_{ij(r)}$s (j = 1, 2), for each i, are actually only one (= 2 – 1) and not two independent effects, since τ_{i1} can be evaluated if τ_{i2} is known and vice versa because model (1) assumes that $\sum\tau = 0$. Similarly τ_k ($k = A, B$) is also one independent parameter. Therefore, total number of independent effects in the model (1) equals

$$p = 1 + \sum_{i=1}^{15} 1 + 1 + \sum_{i=1}^{15} 1$$
$$= 32.$$

S = Unbiased least squares estimator of σ_e^2 [34].

$$S = \frac{1}{N-p} \sum_{r=1}^{16} \sum_{l=1}^{5} \sum_{k=A,B} \left(x_{rkl} - \overline{T} - \sum_{i=1}^{15} \hat{\tau}_{ij(r)} - \hat{\tau}_k - \sum_{i=1}^{15} \hat{\tau}_{m_{ij(r)k}} \right)^2 = \frac{SS_e}{N-p},$$

where SS_e is sum of squares due to error.

If we define $\hat{\tau}_{.h}$ = Estimate of τ_{ij} (factor/interaction), τ_k ('problem') or $\tau_{m_{ij(r)k}}$ (interaction of factor/interaction and problems) at level h, then

$$\hat{\tau}_{.h} = \hat{\mu}_{.h} - \hat{\mu} = \hat{\mu}_{.h} - \overline{T}$$

where $\hat{\mu}_{.h}$ is the *average* of observations under the factor/interaction i (μ_{ij}), problem k (μ_k) or interaction of i and k ($\mu_{m_{ijk}}$) at level h. It is the least squares estimate of respective means. [34]

Let $n_{.h}$ be the number of data points in population under the factor/interaction i, problem k or interaction of i and k (m_{ijk}) at level h. We have made $n_1 = n_2 = n_3 = \ldots\ldots = n$.

The numerator SS_e in S is the *residual sum of squares* or *sum of squares due to error.*

$SS_.$ = Sum of squares of population, where "." corresponds to factor i, problem k or interaction of factors/interactions with problem ($i \times k$).

$SS_.$ is defined as:

$$SS_. = \sum_{j=1,2} n_{.j}(\hat{\mu}_{.j} - \overline{T})^2 = \sum_{j=1,2} n_{.j}\hat{\tau}_{.j}^{\,2}\ .$$

It can be shown that $SS_./\sigma_e^2$ is a χ^2 statistic of degree of freedom 1. Since number of levels of factor i, problem k or interaction of i and k: $i \times k$ is 2, their degrees of freedom, df = 2 – 1 = 1, the df of χ^2 is 1. $(N-p)\ S^2/\sigma_e^2$ is also a χ^2 statistic with df = $N - p$. [34]

$$\therefore \frac{\dfrac{SS_.}{\sigma_e^{\,2}}}{\dfrac{(N-p)S^2}{\sigma_e^{\,2}}} = \frac{SS_.}{(N-p)S^2}$$

is an F-statistic with degrees of freedom: 1, $N-p$. In the present case, $N-p$ = 160 – 32 = 128.

where $\hat{\mu}_{.h}$ is the *average* of observations under the factor/interaction i (μ_{ij}), problem k (μ_k) or interaction of i and k ($\mu_{m_{ijk}}$) at level h. It is the least squares estimate of respective means. [34]

Let $n_{.h}$ be the number of data points in population under the factor/interaction i, problem k or interaction of i and k (m_{ijk}) at level h. We have made $n_1 = n_2 = n_3 = \ldots\ldots\ldots = n$.

The numerator SS_e in S is the *residual sum of squares* or *sum of squares due to error.*

$SS_.$ = Sum of squares of population, where "." corresponds to factor i, problem k or interaction of factors/interactions with problem ($i \times k$).

$SS_.$ is defined as:

$$SS_. = \sum_{j=1,2} n_{.j}(\hat{\mu}_{.j} - \overline{T})^2 = \sum_{j=1,2} n_{.j}\hat{\tau}_{.j}^{\;2} .$$

It can be shown that $SS_./\sigma_e^2$ is a χ^2 statistic of degree of freedom 1. Since number of levels of factor i, problem k or interaction of i and k: $i \times k$ is 2, their degrees of freedom, df = 2 – 1 = 1, the df of χ^2 is 1. $(N-p)\,S^2/\sigma_e^2$ is also a χ^2 statistic with df = $N-p$. [34]

$$\therefore \quad \frac{\dfrac{SS_.}{\sigma_e^{\,2}}}{\dfrac{(N-p)S^2}{\sigma_e^{\,2}}} = \frac{SS_.}{(N-p)S^2}$$

is an F-statistic with degrees of freedom: 1, $N-p$. In the present case, $N-p$ = 160 – 32 = 128.

α, probability of type 1 error (reject the null hypothesis when it is true) for the various hypothesis tested here is .05. The following null hypothesis was tested for all factors/interaction, problem and interactions of factors and problem:

H_0: $\mu_{.1} = \mu_{.2}$

where $\mu_{.j}$ (j = 1, 2) are means of population corresponding to factors/interactions i, problem k or interactions of factors/interactions and problems ($i \times k$) at level j.

In the F-statistic the numerator is:

$$\sum_{j=1,2} n_{.j}(\hat{\mu}_{.j} - \overline{T})^2$$

where $\overline{T} = \dfrac{\hat{\mu}_{.1} + \hat{\mu}_{.2}}{2}$ for two level experiments.

$\therefore$ If $\mu_{.1}$ and $\mu_{.2}$ are equal, $\hat{\mu}_{.1}$ and $\hat{\mu}_{.2}$ will be very close.

Therefore if H_o is true, the value of F-statistic will be relatively small.

$\therefore$ H_0 will be rejected if $F > F_{1,\,128,\,0.95} = 3.92$ where $F_{1,\,128,\,0.95}$ is the onset of the critical region, i.e., the F-distribution quantile corresponding to degrees of freedom 1 in the numerator, 128 in the denominator.

Following formulas have been used to calculate $SS_.$ and SS_e:

In the present case, n = 80 (=16 trials $\times$ 5 replications);

$$SS_{.} = \sum_{j=1,2} n_{.j}(\hat{\mu}_{.j} - \overline{T})^2$$

$$= 80\sum_{j=1,2}(\hat{\mu}_{.j} - \overline{T})^2$$

$$= 80\left(\sum_{j=1,2}\left(\frac{T_{.j}}{80}\right)^2 + 2\overline{T}^2 - 2\overline{T}\sum_{j=1,2}\hat{\mu}_{.j}\right)$$

$$= 80\left(\sum_{j=1,2}\left(\frac{T_{.j}}{80}\right)^2 + 2\left(\frac{T}{2\cdot 80}\right)^2 - 2\left(\frac{T}{2\cdot 80}\right)\left(\frac{T}{80}\right)\right)$$

$$SS_{.} = \frac{1}{80}\sum_{j=1,2} T_{.j} - \frac{T^2}{160}$$

where $T_{.j}$ is total population corresponding to factor/interaction i, problem k or interactions of factors/interactions and problem ($i \times k$) at level j.

For calculating SS_e following identity was used:

$$n \sum_{\substack{\text{All columns,}\\ \text{problems,}\\ \text{interactions}}} \sum_{j=1,2}\left(\hat{\mu}_{.j} - \overline{T}\right)^2 + \sum_{r=1}^{16}\sum_{l=1}^{5}\sum_{k=A,B}\left(x_{rkl} - \overline{T} - \sum_{i=1}^{15}\hat{\tau}_{ij(r)} - \hat{\tau}_k - \sum_{i=1}^{15}\hat{\tau}_{m_{ij(r)k}}\right)^2 = \sum_{r=1}^{16}\sum_{l=1}^{5}\sum_{k=A,B}\left(x_{rkl} - \overline{T}\right)^2$$

$$\Rightarrow \sum_{\substack{\text{All columns,}\\ \text{problems,}\\ \text{interactions}}} SS_{.} + SS_e = SS_T$$

$$\Rightarrow SS_e = \left(\sum_{r=1}^{16}\sum_{l=1}^{5}\sum_{k=A,B} x_{rkl}^2 - \frac{T^2}{160}\right) - \left(80 \sum_{\substack{\text{All columns,}\\ \text{problems,}\\ \text{interactions}}} SS_{.}\right)$$

All those factors and interactions, denoted by i for which $F_i \geq 3.92$ would significant.

4.4 Inferences from parameterisation experiments

Using the experimental data, the "optimum" levels for each of the factors (really GA parameters) are to be determined. This will have to be done for the two scheduling

objectives separately. The "optimum" levels of the significant factors will be determined first. The factors found significant in the parameterisation experiments conducted are listed in Table 6-10 and Table 6-12 for the respective objectives. The factors found insignificant may be set arbitrarily.

It must be expressly stated here that the factors found insignificant here are not necessarily insignificant in reality. They may turn out significant if experiments with more than two levels are conducted as they may have non-linear influence on the algorithm. However, the factors found significant remain so irrespective of whether they have linear or non-linear influence. But still we do not claim that the levels of such factors found in this study are the "most recommended" values for GA implementation. This parameterisation exercise should serve only as an illustration of the procedure.

4.4.1 Makespan as the optimisation objective

The significant factors/interactions are:

1. Population size (column 1 and $1\times P$ where P represents "problem")
2. Probability of crossover (column 2 and $2\times P$)
3. Interaction of population size and percentage elite in selection with the problem ($14\times P = 1\times 15\times P$)

When the interactions were plotted in Figure 4-2 and Figure 4-3, it was found that population size and probability of crossover could remain at the same level (2) in *both* the problems for faster optimisation by the GA. The significance of interaction only means that solution of problem B is somewhat less sensitive to these factors as com-

pared with that of problem *A*, however the effect is still in the *same direction*. Therefore it is recommended that population size and probability of crossover should be set at level 2. Referring to Table 4-1 Factors and their levels chosen in design of experiment., page 68, this implies that

population size (column 1) = 100 (level 2)

probability of crossover (column 2) = 100% (level 2) i.e, "crossover on each solution selected for crossover".

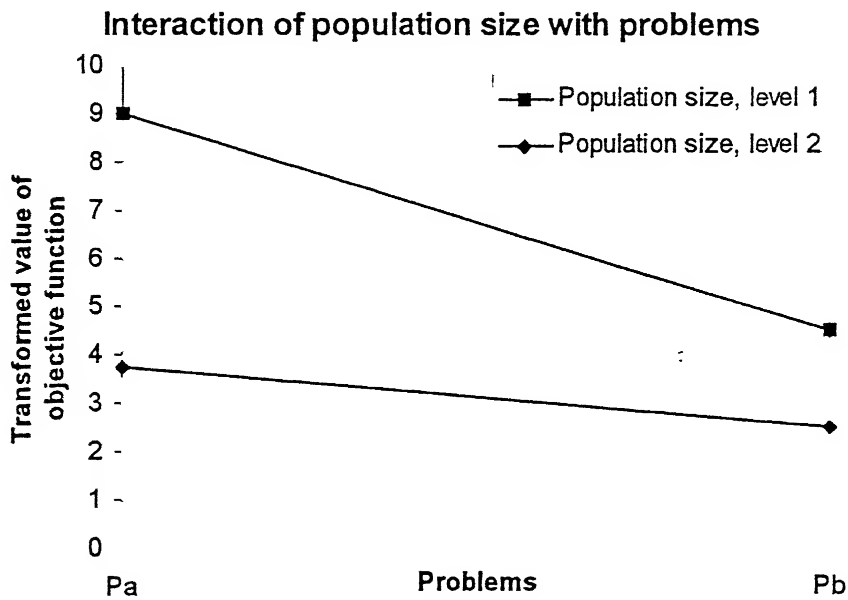


Figure 4-2 Interaction of population size with problems for makespan.

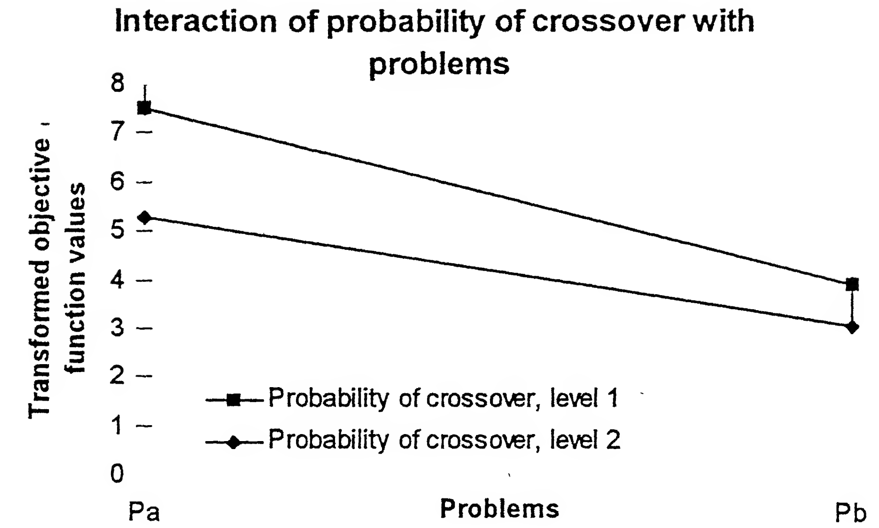


Figure 4-3 Interaction of probability of crossover with problems for makespan.

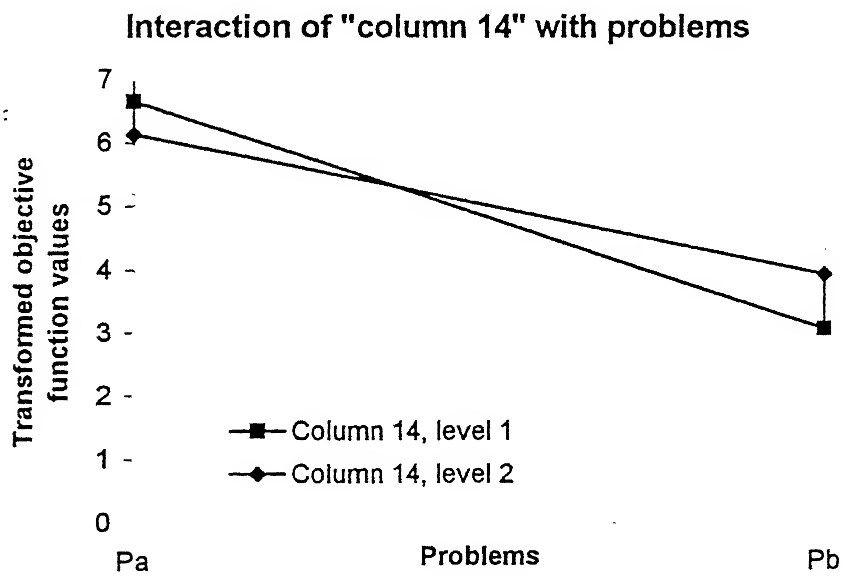


Figure 4-4 Interaction of (column 14) {interaction of population size with elite fraction to be selected} with problems for makespan.

Figure 4-4 shows that interaction (column 14) of population size (column 1) and elite fraction to be selected (column 15) is significant only as an interaction with the prob-

lem and not on its own. That means that averages of the masure of makespan due to both levels of interaction of population size and elite fraction (column 14) will approximately be the same and since the interaction $14 \times P$ is significant, interaction of population size and elite fraction (column 14) will have different effects for both the problems. This is confirmed by Figure 4-4. Therefore for problem A, level 2 of interaction of population size and elite fraction (column 14) should be preferred and for problem B level 1 should be preferred. In L16 OA (Table 4-2) at level 1 of column 14, columns 1 and 15 are at the same levels and at level 2 of column 14, columns 1 and 15 are at different levels. This implies that for better results, columns 1 and 15 should be at different levels for problem A and for problem B, they should be at the same level. Since level of population size (column 1) is already fixed at 100 (2), for problem A elite fraction to be selected should be 10% (level 1) and elite fraction to be selected for problem B should be 50% (level 2).

As the nature of the problem which the algorithm will be solving cannot be predicted, a level independent of the problems will have to be chosen. Therefore, level 1 is chosen based on the comparison of averages under both levels of column 14 in Table 6-9. This implies, at least for the present case, that for better results, population size and elite fraction to be selected must be set at the same level, either both at 1 or both at 2. Since population size is already set at level 2, elite fraction to be selected too should be set at level 2. Therefore,

elite fraction to be selected = 50%

Selection of levels for remaining factors is not of much consequence in this parameterisation, still the averages under both the levels as shown in Table 6-9 are used to set the remaining factors as follows:

type of crossover (column 4) = 2 point crossover (level 2)

probability of mutation (column 8) = 5% (level 1)

4.4.2 Weighted completion time as the optimisation objective

The significant factors/interactions are:

1. population size (column 1)
2. probability of crossover (column 2)
3. interaction of probability of crossover and elite fraction to be selected (column 13)
4. interaction of column 13 with the problem.

Since population size is significant only on its own, without any interaction, its level can be easily fixed. Going by the average effects under both levels of population size in Table 6-11, the population size is set at level 2 i.e.,

population size = 100.

The graph (Figure 4-5) depicting the *interaction* of interaction of probability of crossover and elite fraction to be selected (column 13), and problems, shows that there is not much difference between levels of interaction of probability of crossover and elite fraction to be selected (column 13) for problem *A* while the difference due to it for problem *B* is so large so as to make it significant. Since the average due to level 1 of

interaction of probability of crossover and elite fraction to be selected (column 13) is lower, level 1 is chosen for it. Therefore probability of crossover (column 2) and elite fraction to be selected (column 15) will be at the same level as interaction of probability of crossover and elite fraction to be selected (column 13) represents interaction of probability of crossover (columns 2) and elite fraction to be selected (column 15). Probability of crossover (columns 2) is significant on its own also. Since the average under level 2 of column 2 (see Table 6-11) is lower than that of level 1, level 2 is chosen. This implies that elite fraction to be selected (column 15) too should at level 2. The levels of these factors should be:

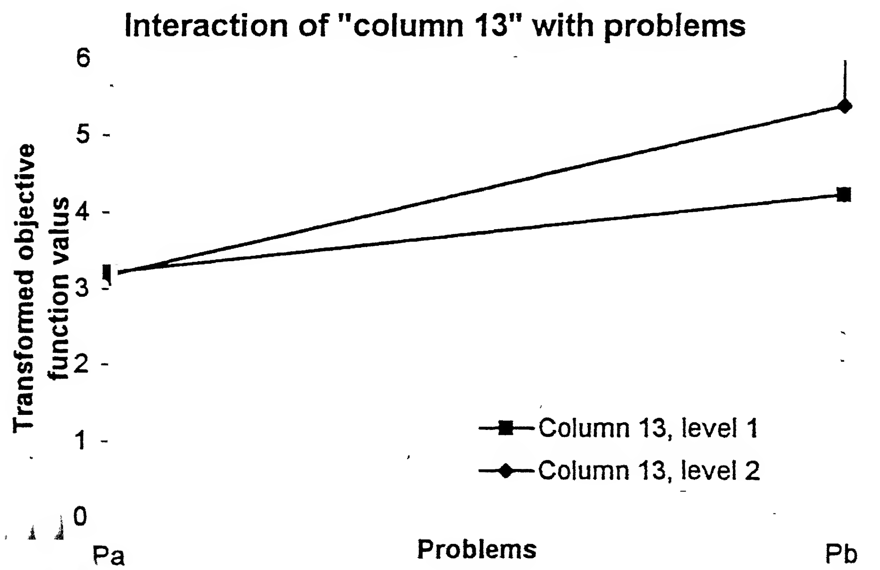
probability of crossover (column 2) = 100% (level 2)

elite fraction to be selected (column 15) = 50% (level 2)

Remaining factors are set as before. They can be set at either levels:

type of crossover (column 4) = 1 point crossover (level 1)

probability of mutation (column 8) = 20% (level 2)



5 Interaction of (column 13) {interaction of population size with type of } with probelms for weighted completion time.

5. Summary and Conclusions

The work on this thesis started with the desire to solve a real life Job Shop problem using optimising methods. One such problem was identified and mathematically formulated. This was the Generalised Job Shop problem. Literature was surveyed for developing insights and familiarity with solution methods, their complexities, requirements on methods and other issues.

Initially the focus was on finding the exact (i.e., the best) solution. Balas' method was considered for possible modification. It soon became apparent that exact analytical method will not be able to go far enough as the problem has too many characterstics which make it difficult to device an appropriate representation to which an efficient analytical and exact method can be applied. The features of the Generalised Job Shop considered in this thesis project are: availability of multiple resources, each operation using one or more than one resource, one or more than one units of each type of resource, release dates, priority, partial ordering and non-availability of resources.

A new method of representing the problem was developed to utilise crossover, mutation and selection operators as used in GA-type heuristics. The feasible solutions were visualised in the form of a sequence of operations which define the order of processing on the resources based on a rule which generates active schedule. Once the algorithm was designed and coded, parameterisation of the GA was attempted and experiments were conducted. It is necessary to do a systematic parameterisation of the GA because [37] with poor parameters GA can be as bad as a random search.

Another experiment was conducted to compare the effectiveness of the GA developed in this project with purely random search. The results comparing the two in case of makespan objective are shown graphically in Figure 5-1 and Figure 5-2; and in case of weighted completion time objective, the comparison is shown in Figure 5-3 and Figure 5-4. Ten thousand random active schedules were generated for each objective. In both set of figures the genetic search produced results better than 3σ (where σ is the standard deviation from the mean of the distribution in Figure 5-1 and Figure 5-3) limit of the random search distribution in *at most* 30 generations i.e., in 3,000 schedule evalution (30×100; 100 is the population size).

Figure 5-1 Distribution of makespan in random search.

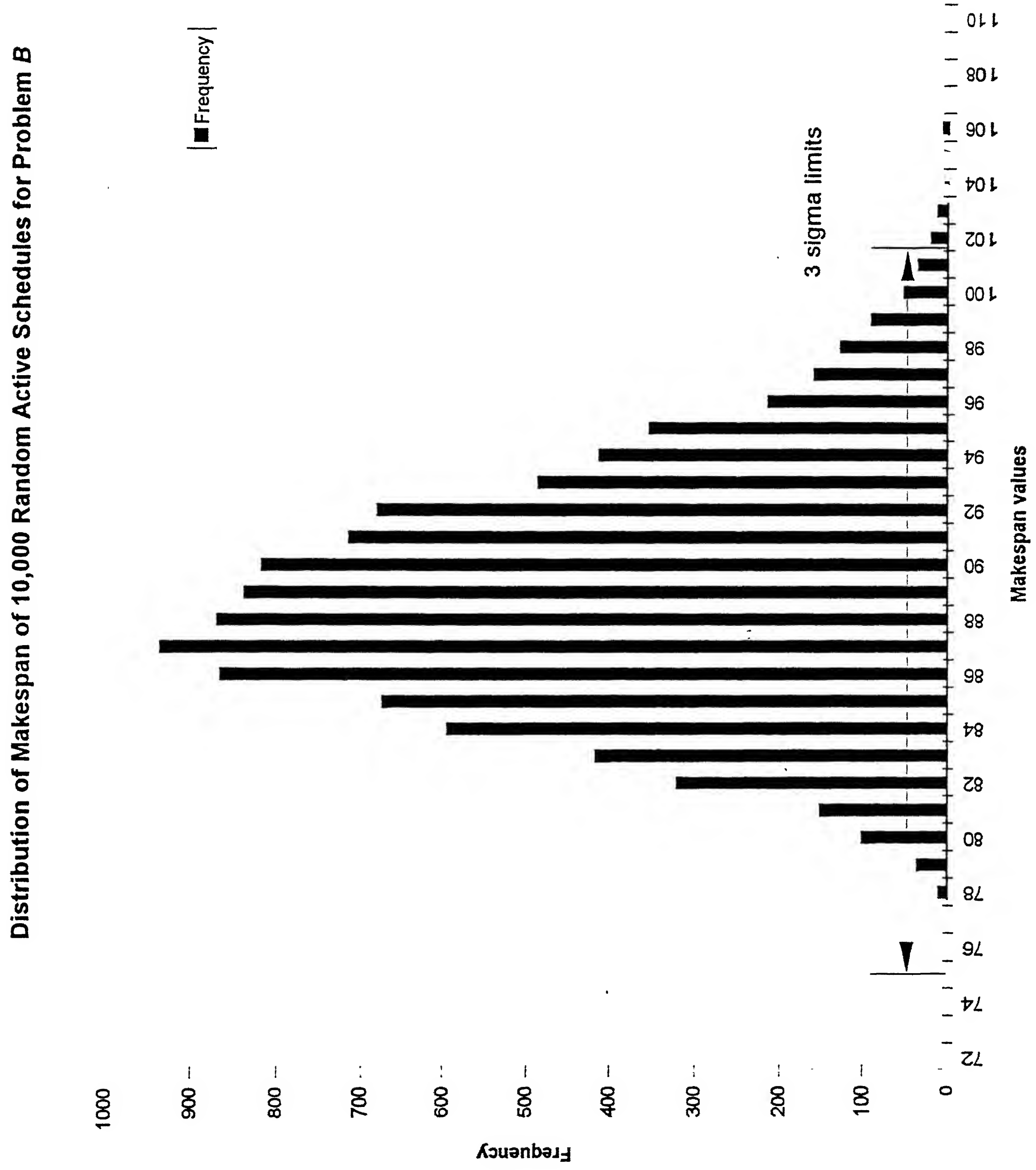


Figure 5-2 Graphical demonstration of performance of the GA for makespan.

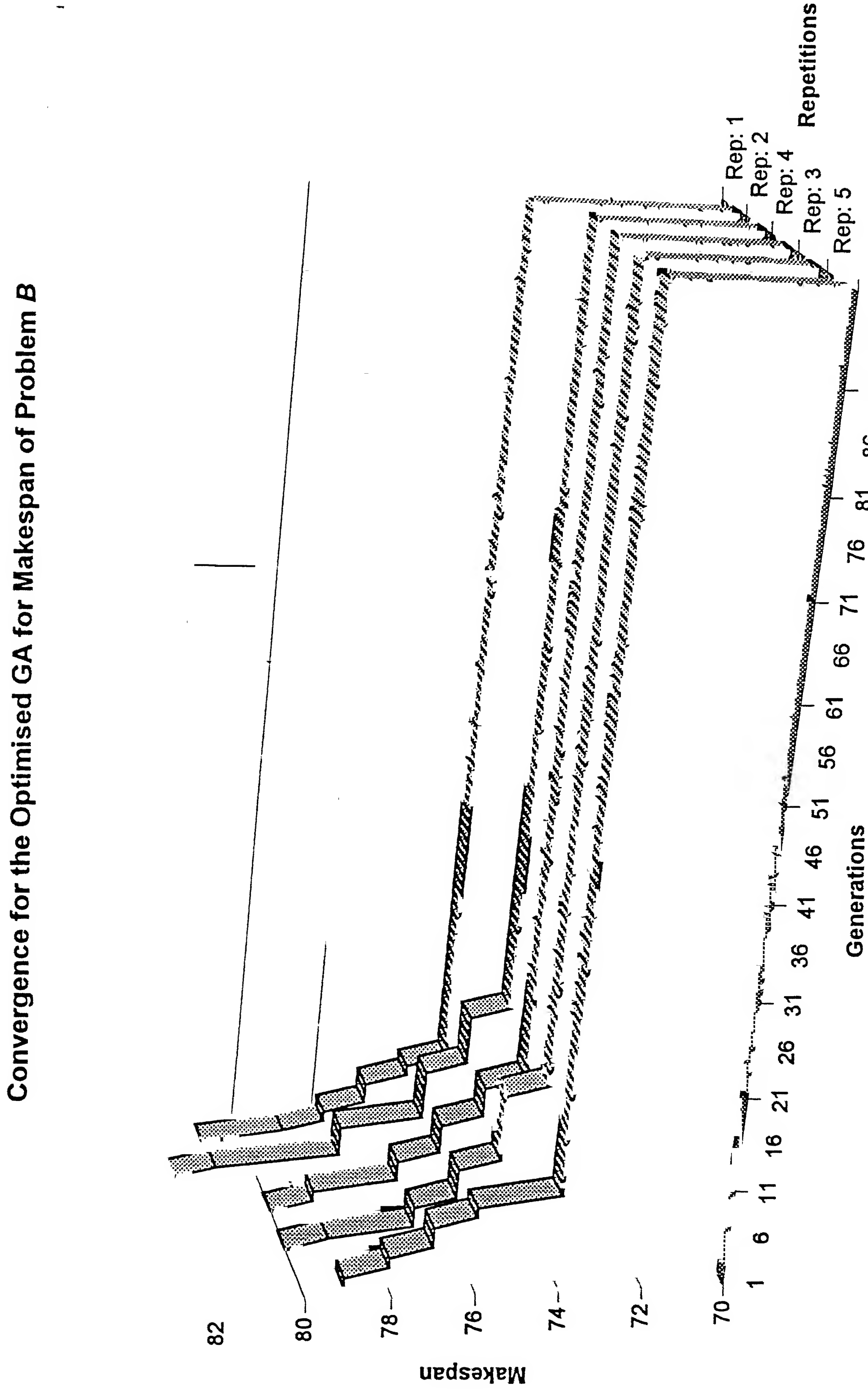


Figure 5-3 Distribution of weighted completion time in random search.

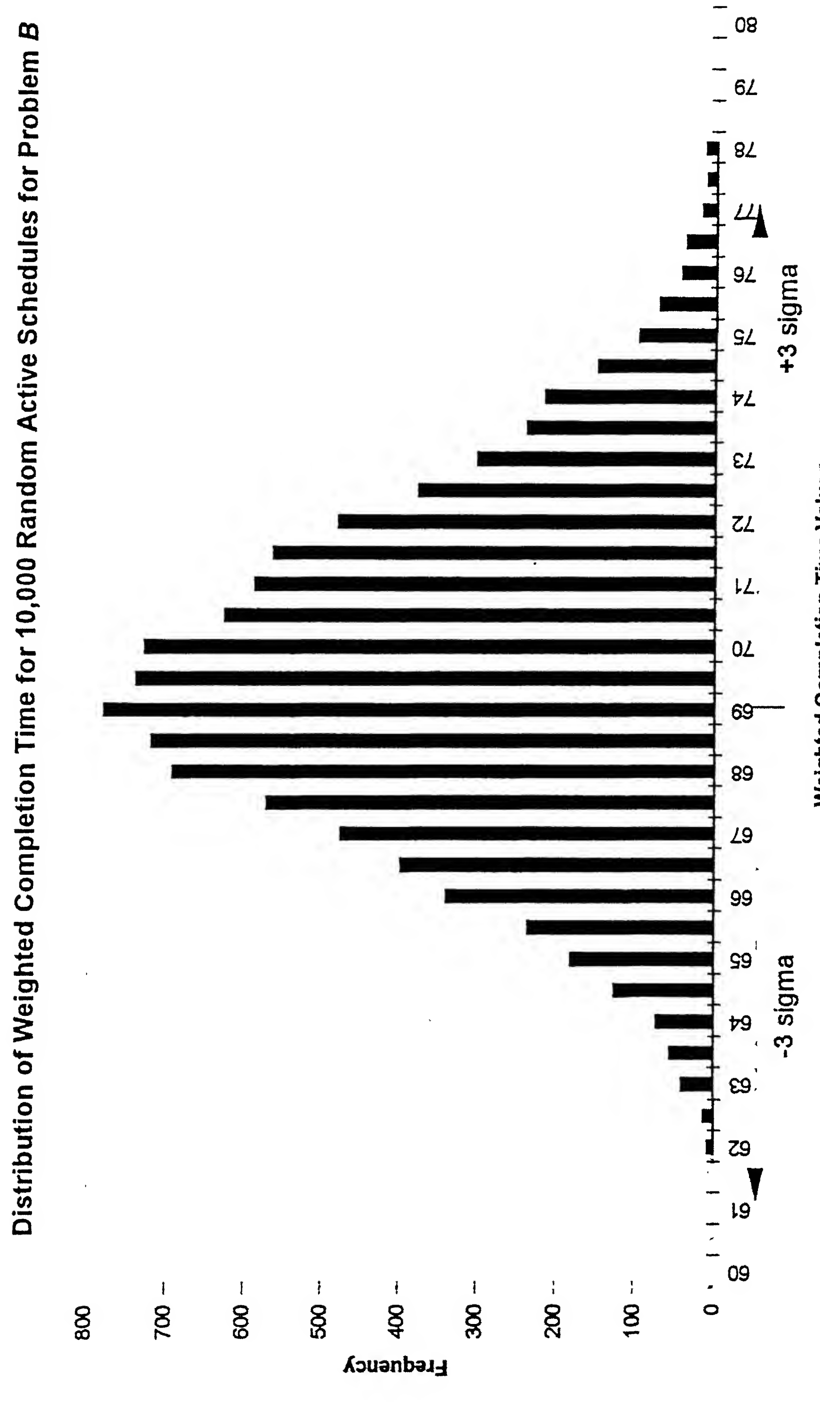
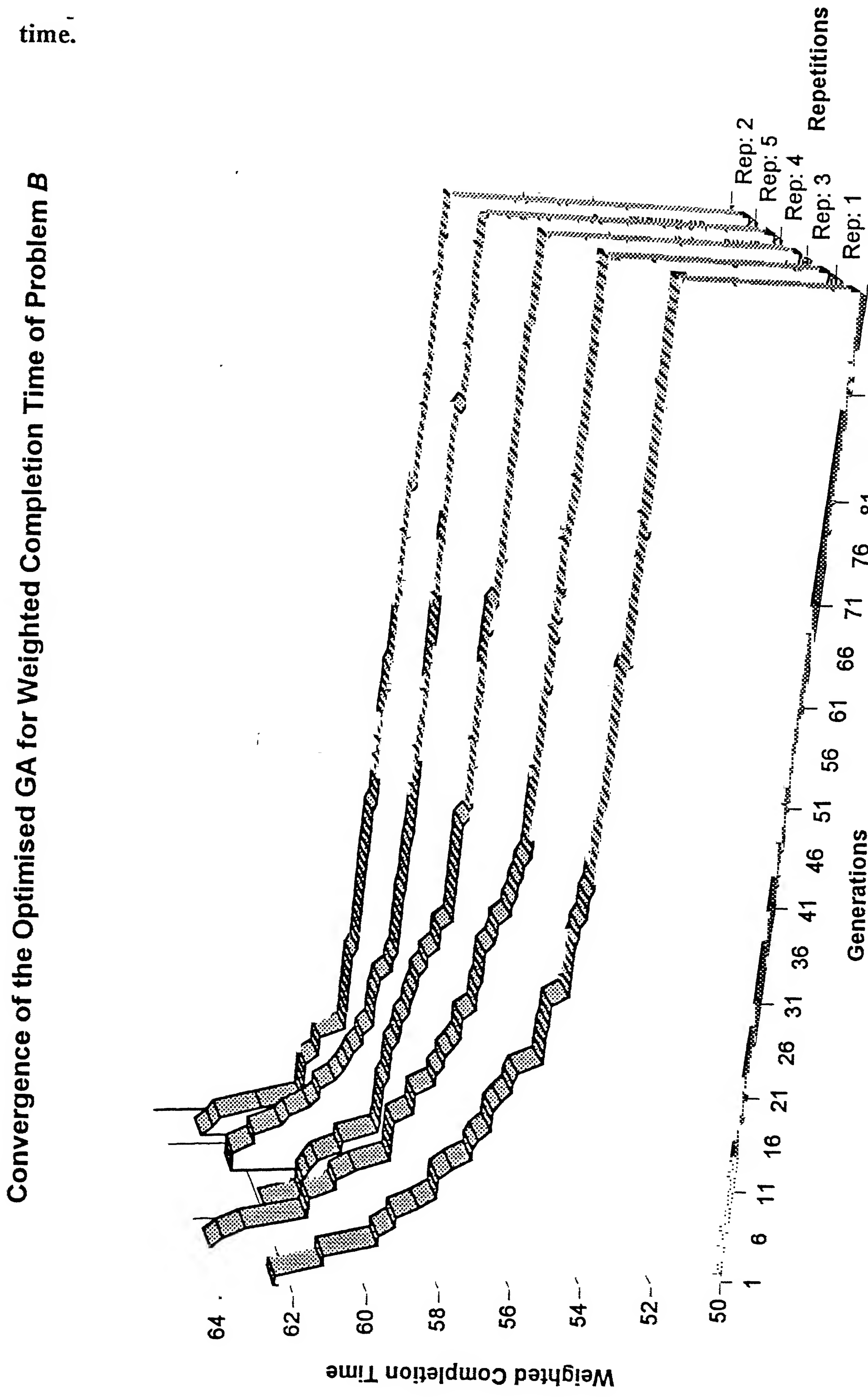


Figure 5-4 Graphical demonstration of performance of GA for weighted completion time.



Thus, the utility of the GA in solving a problem as complex as the GJS has been successfully demonstrated in this work.

5.1 Scope for further work

The work presented in this thesis project may be extended in two directions: the job shop problem dealt with can be made more general, and better and faster GA type search methods may be used to solve the problem.

There are several areas in which this work may be generalised and enhanced. This generalisation should not be very difficult and will serve important purpose. The basic facility for considering multiple resources has been provided in the formulation presented here. It is assumed that the multiple units of a particular type of resource, say lathe machine, are all identical i.e., all of them have same speed. In a more general situation, the processing time may depend upon the particular unit of the resource type chosen for processing. In the other generalisation, the operations may have a choice of resource *types* on which they may be processed, not just choice of the units of a given resource type. For example, such a situation will occur if in a shop, out of 10 milling operators, there are three of them who also know how to operate a lathe machine. Let these 3 operators be called lathe-milling operators. Any turning operation now has a choice

between two *types* of resources: lathe-milling operators and lathe operators. Similarly a milling operation also has choice between lathe-milling operators and milling operators.

There are several ways in which the speed and efficacy of the method may be improved. The simplest among them is parameterisation i.e., by properly adjusting the parameters, significant enhancement in performance may be achieved. The desired parameters may be found through experimentation. Exploration using a two-level parameter experimentation has been attempted here using L16 OA. The effectiveness of experimentation can be increased by having more levels for each parameter and avoiding any confounding of effects, including more parameters and by conducting a full factorial experiment [37].

According to Grefenstette *et al.* [23] and Suh and Gucht [27], the power of any GA may be dramatically improved by incorporating domain information into the recombination operators and by local optimisation. They have demonstrated this in on Traveling Salesman Problem [28] and Sliding Block Problem [29].

Simulated annealing [30], taboo search, gradient descent etc. may be used for local optimisation. Some local optimisation has been attempted in the present work when the *active* schedule is built using the assignment sequence; otherwise building a semiactive schedule would have been far simpler. This can be further improvised using above mentioned approaches. Alternatively, these approaches may be explored without involving genetic algorithms at all.

References

1. "Machine Sequencing Via Disjuctive Graphs: An Implicit Enumeration Algorithm", Egon Balas, Carnegie Mellon University, Pittsburgh, Pennsylvania; Operations Research, 17:941-957, 1969.

2. "Shifting Bottleneck Heuristic", Adam, Balas, Zawack. 1988.

3. "Scheduling: Theory, Algorithms and Systems", Michael Pinedo; Publishers: Prentice Hall, Englewood Cliffs, New Jersey.

4. "Project Scheduling with Resource Constraints", Egon Balas, Carnegie Mellon University, Pennsylvania.

5. "Job Shop Jit: A Case Study", Nicolal R. Phillips; Manufacturing Systems v7 n5 May 1989 p. 48-50.

6. "Flow-shop Setting for Job-shop Scheduling", O. Otolorin; Modelling, Simulation and Control: Environmental, Biomedical, Human & Social Systems v17 n1 1989 p.47-55.

7. “Heuristic N-jobs/M-machines Scheduling Algorithm”, Ali M. Alli; Proceedings of the 11th Annual Conference on Computers and Industrial Engineering v17 n1-4, 1989 p. 359-365.

8. “Job Shop Scheduling by Simulated Annealing”, Peter J.M. Van Laarhoven, H.L. Emile and Jan Karel Lenstra; Operations Research v 40 n1 Jan-Feb 1992 p. 113-125.

9. “Computational Study of the Job Shop Scheduling Problem”, David Applegate and William Cook; ORSA Journal on Computing v3 n2 Spring 1991 p. 149-156.

10. “Expert Systems can do Job Shop Scheduling. An exploration and a proposal.”, John E. Biegel and Lawrence J. Wink; Computers and Industrial Engineering v17 n1-4 1989, p. 347-352.

11. “Knowledge Based Job Shop Scheduling with Controlled Backtrackings”, Odesseus Charalambous and Khalil S. Hindi; computers and Industrial Engineering v24 n3 July, 1993, p.391-400.

12. “Intelligent Disruption Recovery for Decentralised Manufacturing Systems”, Thomas Tsukada And Kang G. Shin; Proceedings–IEE International conference on Robotics and Automation v1. 1993 p. 852-857; Published by IEEE, IEEE service Centre, Piscataway, NJ, USA.

13. "Adapting the Behaviour of a Job Shop Scheduling System", Claude Le Pape and Anne Collinot; Decision Support System v7 n4 Nov 1991 p. 341-35

14. "Simulation Study of Multiproduct Job Shop Scheduling with Machine Failures", R.C. Mishra, P.C. Pandey and J.L.Gaindhar; Journal of Instituition of Engineers (India), Production Engineering Division v69 pt 2 Nov 1988 p 51-54.

15. "Job Shop Scheduling with Alternative Machines", Nabil Nasir and Elsayed; International journal of Production Research v28 n9 Sep 1990 p. 1595-1609.

16. "Job Flow Time in a Job-shop Subject to Machine Breakdowns with Applications in Modelling Job-shop In-process Inventories", Christos P. Koulamas; International Journal of Systems Science v21 n3 Mar 1990 p. 479-486.

17. "Job Shop Scheduling with Tooling Constraints. A Taboo Search Approach", M. Widmer; Journal of Operations Research Society v42 n1 Jan 1991 p. 75-82.

18. "Job-shop Scheduling with Multipurpose Machines", P. Brucker and R. Schlie; University of Osnabrueck, Osnabrueck, Germany. 1990.

19. “A Genetic Algorithm for Job-shop”, E. Falkenauer and S. Bouffouix; Proceedings – IEEE International conference of Robotics and Automation v1 p.824-829. Publishers: IEEE, IEEE Service Centre, Piscataway, NJ, USA.

20. “Genetic Algorithms and JobShop Scheduling”, John E. Biegel and James J. Davern; Computers and Industrial Engineering v19 n1-4 1990. p. 81-91.

21. “Network Flows: Theory, Algorithms, and Applications”, Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin; Publisher: Prentice Hall, New Jersey.

22. “Using Genetic Algorithms to Solve Flow Shop Releases”, Gary A. Cleveland and Stephen F. Smith; The Robotics Institute, Carnegie Mellon University, Pittsburgh.

23. “Genetic Algorithms for Traveling Salesman Problem”, J.J. Grefenstette, R. Gopal, B. J. Romaita, D. Van Gucht; Proceedings an International Conference on Genetic Algorithms and Their Applications, pp. 160-168 (July 1985).

24. “Viewpoints on Optimization”, A. Hertz, D. De-Werra, B. Jaumard, M. Labbe; Publisher: North Holland Publishing Company, Amsterdam (1994).

25. "Performance Evaluation of Genetic Algorithms for Flowshop Scheduling Problems", Tadahiko Murata and Hisao Ishibuchi; Deaprtment of Industrial Engineering. University of Osaka Prefecture.

26. "Taguchi Techniques for Quality Engineering", Phillip J. Ross; Publishers: Mc-Graw Hill Book Company.

27. "Incorporating Heuristic Information Into Genetic Search", Jung Y. Suh, Dirk Van Gucht; Computer Science Department, Indiana University, Bloomington, Indiana.

28. "The Traveling Salesman Problem", E.L. Lawler, J.K. Lenstra, A.G. Rinnooy Kan, D.B. Shmoys; John Wiley & Sons Ltd. (1985).

29. "Principles of Artificial Intelligence", N.J. Nilsson; Tioga Publishnig Company, Palo Alto, Califonia (1980).

30. "Optimisation by Simulated Annealing", S. Kirkpatrick, C.D. Gelatt, M.D. Vecchi; Science Vol. 220 (4598), pp. 671-680 (May 1983).

31. "Adaptation in Natural and Artificial Systems", J. Holland, University of Michigan Press, Ann Arbor (1975).

32. "Genetic Algorithms: What Computers Can Learn from Darwin", Charles T. Walbridge; Technology Review, January 1989.

33. "Conventional Genetic Algorithms for Job Shop Problems", Ryohei Nakano, Takeshi Yamada; Communication and Information Processing Laboratories, NTT, Yokosuka, Japan.

34. "Introduction to Probability Theory and Statistical Inference", Harold J. Larson; Publishers: John Wiley and Sons.

35. "Job Shop Scheduling with Genetic Algorithms", L. Davis; Proceedings of 1st International Conference on Genetic Algorithms and Their Applications (Pittsburgh, PA), pages 136-140 1985.

36. "The C Programming Language", Brian Karnighan, Dennis Ritchie.

37. "An Evaluation of Genetic Algorithms for Robust Design", Krishan Raman; M. Tech. Thesis, IME Department, Indian Institute of Technology, Kanpur, 1993.

38. "Simulation Modelling and Analysis", Averill M. Law, W. David Kelton; McGraw-Hill, Inc 1991.

Appendix A

The analysis programs were used as follows (on the UNIX prompt '$'):

Step 1.

```
$ format problemA.data | gens 40 > problemA.mk
```

This step transformed the raw data in *problemA.data* and selected the data corresponding to 40th generation.

```
$ vi problemA.mk       # To edit out the data
                       # corresponding to weighted
                       # completion
                       # time.
$ step1 problemA.mk > problemA.mk.step1
```

Similarly *problemB.mk.step1* was prepared from *problemB.data*.

```
$ paste problemA.mk.step1 problemB.mk.step1 > mk40.step1
```

The file *mk40.step1* is now having the step 1 table for the makespan objective. Similarly the file *wt40.step1* was prepared having the step 1 table for weighted completion time objective.

Table 6-1 Transformed makespan values for problem *A*.

1	9.43	7.55	11.32	13.21	15.09
2	7.55	15.09	11.32	9.43	7.55
3	9.43	5.66	7.55	7.55	16.98
4	11.32	9.43	11.32	9.43	11.32
5	7.55	7.55	7.55	9.43	7.55
6	11.32	11.32	7.55	9.43	5.66
7	5.66	7.55	3.77	9.43	7.55
8	5.66	5.66	5.66	9.43	7.55
9	5.66	3.77	3.77	5.66	1.89
10	3.77	5.66	5.66	5.66	1.89
11	3.77	3.77	3.77	7.55	5.66
12	3.77	3.77	5.66	5.66	5.66
13	1.89	5.66	1.89	3.77	5.66
14	3.77	5.66	3.77	3.77	1.89
15	1.89	1.89	1.89	3.77	3.77
16	1.89	1.89	0.00	1.89	1.89

Step 1 table from file:
problemA.mk.step1
(Best values)

Table 6-2 Transformed makespan values for problem *B*.

1	4.17	5.56	2.78	4.17	2.78
2	4.17	4.17	2.78	8.33	6.94
3	6.94	6.94	11.11	5.56	2.78
4	5.56	5.56	4.17	4.17	4.17
5	1.39	2.78	4.17	5.56	1.39
6	2.78	2.78	5.56	4.17	4.17
7	2.78	4.17	4.17	4.17	4.17
8	5.56	4.17	5.56	4.17	2.78
9	4.17	0.00	2.78	2.78	2.78
10	4.17	5.56	2.78	4.17	4.17
11	4.17	1.39	2.78	2.78	1.39
12	0.00	1.39	1.39	2.78	2.78
13	2.78	2.78	2.78	0.00	2.78
14	2.78	2.78	2.78	0.00	2.78
15	2.78	0.00	2.78	1.39	1.39
16	2.78	2.78	4.17	2.78	2.78

Step 1 table from file:
problemB.mk.step1
(Best values)

Table 6-3 Table of transformed makespan values of problem *A* and problem *B* merged together.

1	9.43	7.55	11.32	13.21	15.09	4.17	5.56	2.78	4.17	2.78
2	7.55	15.09	11.32	9.43	7.55	4.17	4.17	2.78	8.33	6.94
3	9.43	5.66	7.55	7.55	16.98	6.94	6.94	11.11	5.56	2.78
4	11.32	9.43	11.32	9.43	11.32	5.56	5.56	4.17	4.17	4.17
5	7.55	7.55	7.55	9.43	7.55	1.39	2.78	4.17	5.56	1.39
6	11.32	11.32	7.55	9.43	5.66	2.78	2.78	5.56	4.17	4.17
7	5.66	7.55	3.77	9.43	7.55	2.78	4.17	4.17	4.17	4.17
8	5.66	5.66	5.66	9.43	7.55	5.56	4.17	5.56	4.17	2.78
9	5.66	3.77	3.77	5.66	1.89	4.17	0.00	2.78	2.78	2.78
10	3.77	5.66	5.66	5.66	1.89	4.17	5.56	2.78	4.17	4.17
11	3.77	3.77	3.77	7.55	5.66	4.17	1.39	2.78	2.78	1.39
12	3.77	3.77	5.66	5.66	5.66	0.00	1.39	1.39	2.78	2.78
13	1.89	5.66	1.89	3.77	5.66	2.78	2.78	2.78	0.00	2.78
14	3.77	5.66	3.77	3.77	1.89	2.78	2.78	2.78	0.00	2.78
15	1.89	1.89	1.89	3.77	3.77	2.78	0.00	2.78	1.39	1.39
16	1.89	1.89	0.00	1.89	1.89	2.78	2.78	4.17	2.78	2.78

Step 1 table from file:
problemA.mk.step1
(Best values)

Step 1 table from file:
problemB.mk.step1
(Best values)

Table 6-4 Table of transformed weighted completion time values of problems *A* and problem *B* merged together.

17	5.58	8.15	2.18	3.53	5.97	8.45	8.41	7.16	5.80	3.99
18	6.54	2.18	2.03	4.08	5.17	9.04	7.34	7.95	7.25	8.68
19	10.48	6.93	5.30	3.95	5.17	10.65	9.83	10.69	7.93	6.89
20	5.45	4.75	2.86	10.90	2.99	3.13	7.43	8.63	5.64	7.62
21	4.62	8.44	2.18	4.36	3.66	6.46	4.81	6.14	2.72	4.05
22	3.27	6.26	2.03	3.27	6.26	3.94	4.87	10.43	12.49	7.02
23	2.70	6.26	4.36	4.49	6.39	7.07	8.86	5.48	7.71	7.16
24	5.30	8.57	2.03	3.40	4.75	6.28	5.75	4.01	5.48	5.73
25	3.27	1.90	1.61	0.00	2.03	3.35	4.40	0.82	3.76	3.20
26	3.27	1.77	0.00	1.90	1.09	1.93	5.28	0.47	0.61	2.81
27	1.61	3.12	0.00	0.68	2.03	2.15	4.26	4.87	1.79	2.67
28	0.00	0.00	0.00	1.77	1.48	4.31	4.92	2.38	2.95	1.79
29	2.31	1.09	1.77	1.61	1.90	3.45	1.97	0.68	1.75	1.32
30	1.77	1.48	0.81	2.18	1.35	3.08	2.90	0.00	2.26	1.99
31	1.48	1.90	0.00	1.61	0.68	1.38	6.98	0.09	3.90	1.41
32	2.18	1.35	0.81	2.03	2.70	2.88	1.11	3.72	3.69	1.52

Step 1 table from file:
problemA.wt.step1
(Best values)

Step 1 table from file:
problemB.wt.step1
(Best values)

Step 2.

```
$ step2 < mk40.step1 > mk40.step2
```

Table 6-5 Table of step 2 values for makespan.

1	56.60	19.46
2	50.94	26.39
3	47.17	33.33
4	52.82	23.63
5	39.63	15.29
6	45.28	19.46
7	33.96	19.46
8	33.96	22.24
9	20.75	12.51
10	22.64	20.85
11	24.52	12.51
12	24.52	8.34
13	18.87	11.12
14	18.86	11.12
15	13.21	8.34
16	7.56	15.29
S:	511.29	279.34

Table 6-6 Table of step 2 values for weighted completion time.

17	25.41	33.81
18	20.00	40.26
19	31.83	45.99
20	26.95	32.45
21	23.26	24.18
22	21.09	38.75
23	24.20	36.28
24	24.05	27.25
25	8.81	15.53
26	8.03	11.10
27	7.44	15.74
28	3.25	16.35
29	8.68	9.17
30	7.59	10.23
31	5.67	13.76
32	9.07	12.92
S:	25[illegible].33	383.77

Step 4.

```
$ ANOVA mk40.pr mk40.sec mk40.ter mk40.an
```

The ANOVA table is in the file *mk40.an*. See Table 6-9 ANOVA Table of Makespan data.

This has been done using the program filter as follows:

```
$ filter mk40.an 3.92
```

The results are shown in the Table 6-10.

Table 6-9 ANOVA Table of Makespan data.

1	520.60	1	6.75	3.14	160.99
2	95.06	1	5.71	4.17	29.40
3	9.40	1	5.18	4.70	2.91
4	5.22	1	5.12	4.76	1.61
5	1.59	1	4.84	5.04	0.49
6	3.11	1	4.80	5.08	0.96
7	0.15	1	4.91	4.97	0.05
8	1.84	1	4.83	5.05	0.57
9	0.04	1	4.93	4.96	0.01
10	0.70	1	5.01	4.88	0.22
11	2.54	1	5.07	4.82	0.79
12	4.05	1	4.78	5.10	1.25
13	0.00	1	4.95	4.94	0.00
14	1.22	1	4.85	5.03	0.38
15	1.90	1	5.05	4.83	0.59
S:	647.42	15			
OP:	336.26	1			103.99
1xP	106.02	1			32.79
2xP	18.18	1			5.62
3xP	0.05	1			0.02
4xP	11.44	1			3.54
5xP	11.24	1			3.48
6xP	10.91	1			3.37
7xP	6.09	1			1.88
8xP	1.12	1			0.35
9xP	1.66	1			0.51
10xP	1.29	1			0.40
11xP	0.24	1			0.07
12xP	2.94	1			0.91
13xP	4.11	1			1.27
14xP	19.19	1			5.93
15xP	0.50	1			0.15
ST2:	1178.66	31			
e	413.91	128			
ST3:	1592.57	159			

Table 6-10 The list of columns with *F*-values more than 3.92 (95%, 1, 128).

col	SS	df	level 1	level 2	F
1	520.60	1	6.75	3.14	160.99
2	95.06	1	5.71	4.17	29.40
OP:	336.26	1			103.99
1xP	106.02	1			32.79
2xP	18.18	1			5.62
14xP	19.19	1			5.93

Table 6-11 ANOVA Table of weighted completion time data.

col	SS	df	level 1	level 2	F
1	610.04	1	5.95	2.04	193.62
2	13.69	1	4.29	3.70	4.34
3	5.07	1	4.17	3.82	1.61
4	4.66	1	3.82	4.17	1.48
5	1.84	1	3.89	4.10	0.58
6	0.29	1	3.95	4.04	0.09
7	2.42	1	3.87	4.12	0.77
8	2.61	1	4.12	3.87	0.83
9	0.39	1	4.04	3.95	0.12
10	6.37	1	4.19	3.79	2.02
11	0.54	1	4.05	3.94	0.17
12	8.47	1	3.76	4.22	2.69
13	12.80	1	3.71	4.28	4.06
14	0.01	1	4.00	3.99	0.00
15	0.06	1	4.01	3.98	0.02
S:	669.26	15			
OP:	103.11	1			32.73
1xP	8.06	1			2.56
2xP	5.84	1			1.85
3xP	0.01	1			0.00
4xP	0.41	1			0.13
5xP	6.69	1			2.12
6xP	0.05	1			0.02
7xP	0.45	1			0.14
8xP	0.63	1			0.20
9xP	0.90	1			0.29
10xP	0.01	1			0.00
11xP	0.38	1			0.12
12xP	12.14	1			3.85
13xP	14.52	1			4.61
14xP	2.48	1			0.79
15xP	0.59	1			0.19
ST2:	825.53	31			
e	403.30	128			
ST3:	1228.83	159			

Table 6-12 The list of columns with *F*-values more than 3.92 (95%, 1, 128).

col	SS	df	level 1	level 2	F
1	610.04	1	5.95	2.04	193.62
2	13.69	1	4.29	3.70	4.34
13	12.80	1	3.71	4.28	4.06
OP:	103.11	1			32.73
13xP	14.52	1			4.61

Appendix B

The C program used to implement the Genetic Algorithm for solving generalised job shop scheduling problem is presented here. A flowchart describing the outline of this program is provided at the end of the program.

```
/* This program is C implementation of Genetic Algorithm used for solving      */
/* Generalised Job Shop Scheduling problem.  The data is taken from three      */
/* sources: Data about the resources (file: Resources), data about products    */
/* (file: Products), and data about program variables (keyboard).  Output      */
/* is produced in two forms: Best schedule in the same format as the data      */
/* in Resources (file: BestSch), Gantt chart of any schedule in the last       */
/* generation.  GA statistics is also produced as the program is running       */
/* (standard output).                                                          */

/* This program was written by Ankur Bhatnagar; M. Tech. (IME, IIT/K), 1996   */
/* as part of thesis project under the guidence of Dr. Tapan P. Bagchi. The   */
/* algorithm was also devised as part of the project.                         */


#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define SIZE 30          /* SIZE of char arrays; e.g., Resource types etc. */
#define EPS 0.0001       /* EPSillon: A very small number; used for comparing
                         equalities and inequalities of floating point no.s */

/* Data structures */

typedef struct res_tag {
      char Resource [SIZE];     /* Type of resource */
      int No;                   /* No. of resources used of the above type */
      struct res_tag *suc;
} resource;                   /* List of resource types used in an operation */

typedef struct oplist_tag {
      char ProductId [SIZE];    /* Product identification (name) */
      int OperationNo;          /* The identification of an operation on the
                               product as the sequence number.
                                                                  */

      struct operation_tag {
            char ProductId [SIZE], Schd; /* Product identification; Schd: A flag
                                whether this operation has been
                                scheduled.                                    */
            float ProcTime, Margin;  /* Processing time and margin */
            resource *ResList;       /* List of resources used in the operation */
            struct oplist_tag *Pred, *Succ;
            int OperationNo;         /* Operation identification no. as the
```

```
                                            sequence of processing on the product. */
            struct operation_tag *pre, *suc;
        } *Operation;          /* Pointer (ptr) to structure of an operation. */

        int Scheduled;         /* A flag to identify whether the operation is
                                  scheduled. */
        struct oplist_tag *suc;
} oplist;  /* List of operations. */

typedef struct operation_tag operation;

typedef struct date_tag {
        int Day, Month, Year;
} date;

typedef struct prod_tag {
        char ProductId [SIZE];
        operation *First, *Starting;  /* First operation on the product; The
                                         operation for which the scheduling will
                                         start on the product (first unscheduled
                                         operation of the product). */
        int Priority;
        date ReleaseDt;   /* Release Date */
        struct prod_tag *suc;
} product;   /* Structure of a product. */

typedef struct opseq_tag {
        operation * Operation;
        struct opseq_tag *pre, *suc;
} opseq;   /* Sequence of operations. */

typedef struct box_tag {
        float StartTime;           /* The time at which processing starts */
        operation * Operation;     /* On this operation. */
        struct box_tag *pre, *suc;
} box;                      /* A box representing an operation on a Gantt chart */

typedef struct ressch_tag {
        char ResourceType [SIZE], ResourceId  [SIZE]; /* Type of resource and

identification of a

particular resource. */
        box *JobSeq, *JobSeqR;  /* Start of the operations on the resource (first
                                   operation on a resource in a Gantt chart); Last
                                   operation on the resource.  Both ptr.s point to
                                   different ends of a sequence of boxes in the
                                   Gantt chart in front of the resource. */
        struct ressch_tag *suc;
} ressch;  /* Structure of a resource in a schedule */

typedef struct schedule_tag {
        ressch *Resources;   /* Ptr. to a list of resources. */
        opseq *AssgnmtSeq;   /* Ptr. to assignment sequence. */
        date StartingDt;     /* The date when the schedule will come in force. */
        float Obj;           /* Objective function value of the schedule. */
} schedule;                  /* Structure of a schedule. */

void Read_ProductFile();
void Read_ResourceFile();
void Read_ProgVariables();
void CreateFirstGeneration();
void EvaluatePopulation();
void Crossover();
```

```
void Sele
void OutputSchedules();
void error(char *, char *);
void add_opseq (opseq **listp, opseq *element);
void add_ressch (ressch **listp, ressch *element);
void add_product (product **listp, product *element);
void add_oplist(oplist **listp, oplist *element);
void add_resource(resource **listp, resource *element);
void adde_opseq (opseq **listp, opseq *element);
void adde_box (box **listp, box *element);
void adde_operation (operation **listp, operation *element);
void adde_box (oplist *key, box *list);
box *search_box (opseq *key, opseq *list);
opseq *search_opseq (int key, operation *list);
operation *search_operation (char *key, product *list);
product *search_product (opseq **listp, opseq *element);
int delete_opseq (box *link, box *element, int pos);
void insert_box (const schedule *, const schedule *);
int schcmp(int a, int b, int *n);
void xover(int *);
void OneSelection(int p);
void CreateAssgnmtSeq(int p);
void AssgnTheSeq(schedule);
void WtdCompletion(schedule);
float Makespan(schedule);
float schedulable ( operation *Op, int p );
int schedulable_op_in_opseq (operation *key, opseq *list);
opseq *search_op_in_opseq (date Dt);
float TimeSince1996Till (int day);
float DayToTime( date Rel, date RSchDt );
float DtToTime (float t);
float TimeToDt (float t);
date TimeToDay (int y);
int leap (int y);
int NoOfDays(int y);
int WriteSch (int i, FILE *f);
void random (int l);
int schcpy( schedule *To, const schedule *From);
void FinishTime(box *);
float ProcessTime(operation *);
float search_op_in_box( operation *, box *);
box *search_op_in_box( operation *, box *);
void gantt (schedule, int);
void deletesch (int);
void delete_ressch (ressch **listp, ressch *element);
int delete_box (box **listp, box *element);
int swap (schedule *, schedule *);
void twoptxover (int, int, int *);
void mutn (opseq **, int);
int Mutation (int);
int Mutation (int);

/* Global Variables */

static char daytab[2][13] = {
        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
        {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}
};  /* No. of days in various months in ordinary year and leap year. */

product *ProdList = NULL;
int Pop,        /* Population size. */
        TerminatingCond, Generation = 0,
        NGen,      /* No. of generations */
        NoProducts = 0,     /* No. of products. */
        NoOperations = 0, BestSchd,      /* Array element having the best schedule in
                                            the array of schedules. */
        TotalPopSize;  /* Total population size after the daughter solutions are
                          produced after crossover. */
```

```
schedule *Sch,    /* Array of schedules. */
     *RSch;
float Best, Avg, BestLast, AvgLast, /* Best value, Average value, Best value i
                                       the last generation and Average value i
                                       the last generation of the objective
                                       function. */

     time_taken,        /* Processing time taken by the program. */
     unit = 1,          /* Time unit in terms of days. */
     prob_mutation,     /* Probability of mutation. */
     prob_xover,        /* Probability of crossover. */
     elite,             /* Elite fraction to be selected. */
     OFn,               /* Type of objective fubction. */
     x_type;            /* Type of crossover. */

main()
{
     Read_ProductFile();
     Read_ResourceFile();
     Read_ProgVariables();

     Sch = ( schedule *) calloc(3*Pop, sizeof( schedule ));

     time_taken = clock()/((float) CLOCKS_PER_SEC);
     CreateFirstGeneration();

     EvaluatePopulation();

     while (1) {
          Crossover();

          EvaluatePopulation();

          TerminatingCond = Generation > NGen;
          if (TerminatingCond) break;

          Selection();
     }
     time_taken = clock()/((float) CLOCKS_PER_SEC);

     printf("\nTime taken by the processor: %f sec.\n", time_taken);

     qsort (Sch, 3*Pop, sizeof( schedule ),
               (int (*)(const void *, const void *)) schcmp);
     OutputSchedules();
}


void Read_ProductFile()
{
     FILE *prodf;
     int n;
     char str [SIZE], str1 [SIZE], ans [8];
     product *Prodp, *Prodp1;
     operation *Opp;
     resource *Resp;
     oplist *Oplstp;

     prodf = fopen("Products", "r");
     if (prodf == NULL) error("cannot open: ", "Products");

     fscanf(prodf, "%s", str);

     while( feof(prodf) == 0 )  {
```

```
if (strcmp(str, "Product_id:") != 0)
        error("file format incorrect for product", "");

Prodp = ( product * ) calloc( 1, sizeof(product) );
++ NoProducts;

n = fscanf( prodf, "%s%*s%d%*s%d%d%d", Prodp->ProductId,
        & Prodp->Priority, & Prodp->ReleaseDt.Day,
        & Prodp->ReleaseDt.Month, & Prodp->ReleaseDt.Year);
if (n != 5) error("info incorrectly stored for product ",
              Prodp->ProductId);

fscanf( prodf, "%s", str);
if (strcmp(str, "Operation_no.:") != 0)
        error("no operation : ", Prodp->ProductId);

do  {
        Opp = ( operation * ) calloc(1, sizeof( operation ));
        fscanf( prodf, "%d%*s%s%*s%f%*s%f%*s%s",
        & Opp->OperationNo, ans, & Opp->ProcTime, &Opp->Margin,
        str);
        strcpy(Opp->ProductId, Prodp->ProductId);

        if (toupper(ans[0]) == 'Y') Opp->Schd = 'Y';
        else Opp->Schd = 'N';

        while( strcmp(str, "Preceding_operations:") != 0)  {
                Resp = (resource *) calloc(1, sizeof(resource));
                strcpy( Resp->Resource, str);
                fscanf(prodf, "%d%s", & Resp->No, str);
                add_resource (& Opp->ResList, Resp);
        }
        fscanf(prodf, "%s", str);

        while( strcmp(str, "Succeeding_operations:") != 0)  {
                Oplstp = ( oplist * ) calloc(1, sizeof(oplist));
                strcpy(Oplstp->ProductId, str);
                fscanf(prodf, "%d%s", & Oplstp->OperationNo,
                                          str);
                add_oplist (& Opp->Pred, Oplstp);
        }
        fscanf(prodf, "%s", str);

        while(strcmp(str, "Operation_no.:") != 0 &&
        strcmp(str, "Product_id:") != 0 && feof(prodf) == 0) {
                Oplstp = (oplist *) calloc(1, sizeof(oplist));
                strcpy(Oplstp->ProductId, str);
                fscanf(prodf, "%d%s", & Oplstp->OperationNo,
                                          str);
                add_oplist (& Opp->Succ, Oplstp);
        }

        adde_operation ( & Prodp->Starting, Opp);
        if (toupper(ans[0]) == 'N' && Prodp->First == NULL)
                Prodp->First = Opp;

        if (Prodp->First != NULL) ++ NoOperations;
        if (Opp->pre != NULL)  {
                Oplstp = (oplist *) calloc(1, sizeof(oplist));
                strcpy(Oplstp->ProductId, Opp->pre->ProductId);
                Oplstp->OperationNo = Opp->pre->OperationNo;
                Oplstp->Operation = Opp->pre;
                add_oplist(&Opp->Pred, Oplstp);
```

```
                    Oplstp = (oplist     *) calloc(1, sizeof(oplist));
                    strcpy(Oplstp->ProductId, Opp->ProductId);
                    Oplstp->OperationNo = Opp->OperationNo;
                    Oplstp->Operation = Opp;
                    add_oplist(Opp->pre->Succ, Oplstp);
                }
            } while (strcmp(str, "Product_id:") != 0 && feof(prodf) == 0);

            add_product (&ProdList, Prodp);
        }
        fclose(prodf);

        /* Resolving the predecessors and successors of the operations */

        for ( Prodp = ProdList ; Prodp ; Prodp = Prodp->suc )
          for (Opp=Prodp->Starting; Opp   ; Opp = Opp->suc)  {
            for ( Oplstp = Opp->Pred ; Oplstp ; Oplstp = Oplstp->suc )  {

                Prodp1 = search_product (Oplstp->ProductId, ProdList);
                if ( Prodp1 == NULL )
                    error(Prodp->ProductId, ": cannot resolve its "
                                            "predecessor");
                Oplstp->Operation = search_operation(Oplstp->OperationNo
                                          , Prodp1->Starting);
                if ( Oplstp->Operation == NULL )
                    error( Prodp1->ProductId, ": cannot resolve "
                                "its Operation no. (Pred)");
            }
            for ( Oplstp = Opp->Succ ; Oplstp ; Oplstp = Oplstp->suc )  {

                Prodp1 = search_product (Oplstp->ProductId, ProdList);
                if ( Prodp1 == NULL )
                    error(Prodp->ProductId, ": cannot resolve its "
                                            "successor");
                Oplstp->Operation = search_operation(Oplstp->OperationNo
                                          , Prodp1->Starting);
                if ( Oplstp->Operation == NULL )
                    error( Prodp1->ProductId, ": cannot resolve "
                                "its Operation no. (Succ)");

            }
          }
}

void Read_ResourceFile()
{
        FILE *resf;
        int n, OpNo;
        char str [SIZE], str1 [SIZE], ams [8];
        product *Prodp, *Prodp1;
        operation *Opp;
        resource *Resp;
        oplist *Oplstp;
        ressch *Resschp;
        box *Boxp;

        resf = fopen("Resources", "r");
        if (resf == NULL) error("cannot  open: ", "Resources");

        RSch = ( schedule * ) calloc(1,  sizeof( schedule ));
        fscanf(resf, "%*s%d%d%d", &RSch->StartingDt.Day,
                    &RSch->StartingDt.Month, &RSch->StartingDt.Year);
        fscanf(resf, "%s", str);

        while (feof(resf) == 0 && strcmp(str, "****") != 0)  {
```

```
        if (strcmp(str, "Resource_type:") != 0)
            error("file format incorrect for resource schedule","");

        Resschp = ( ressch * ) calloc(1, sizeof(ressch));
        fscanf(resf, "%s%*s%s%s", Resschp->ResourceType,
                                Resschp->ResourceId, str);

        while (strcmp(str, "Resource_type:") != 0 && feof(resf) == 0
                                && strcmp(str, "****") != 0)  {
            fscanf(resf, "%s", str1);

            Boxp = ( box * ) calloc(1, sizeof(box));
            if ( strcmp(str1, "Down") != 0 )  {
                fscanf(resf, "%*s%d%*s%f%s", &OpNo,
                                &Boxp->StartTime, str);
                Prodp = search_product (str1, ProdList);
                if ( Prodp == NULL )
                    error(str1, " product in res file does"
                            "not exist in prod file");
                Boxp->Operation= search_operation(OpNo,
                                Prodp->Starting);
                if (Boxp->Operation == NULL)
                    error(str1, ": an operation on this "
                            "product wasn't found in prod");

                adde_box ( & Resschp->JobSeq, Boxp);
                continue;
            }

            Opp = ( operation * ) calloc(1, sizeof( operation ));
            strcpy(Opp->ProductId, str1);
            fscanf(resf, "%*s%s%*s%f%*s%f%s", str1,
                    &Boxp->StartTime, &Opp->ProcTime, str);
            strcat(Opp->ProductId, str1);
            Boxp->Operation = Opp;
            adde_box ( & Resschp->JobSeq, Boxp);
        }
        Resschp->JobSeqR = Boxp;
        add_ressch (& RSch->Resources, Resschp);
    }
    fclose(resf);
}


void Read_ProgVariables()
{
    printf("Enter the population: ");
    scanf("%d", &Pop);
    printf("No. of generations: ");
    scanf("%d", &NGen);
    printf("Probability of crossover: ");
    scanf("%f", &prob_xover);
    printf("Type of crossover ? (1. One pt. crossover, 2. Two pt. "
    "crossover) ");
    scanf("%f", &x_type);
    if (x_type + EPS < 1 || x_type > 2 + EPS)
        error("bad choice of xover type", "");
    -- x_type;
    x_type *= 100;
    printf("Probability of mutation: ");
    scanf("%f", &prob_mutation);
    printf("%%age of elite to be retained while selection: ");
    scanf("%f", &elite);
    elite = (elite * Pop)/100;
```

```
	printf("Which Obj. function ? (1. Makespan, 2. Wtd Completion) ");
	scanf("%f", &OFn);
	if (OFn + EPS < 1 || OFn > 2 + EPS)
		error("bad choice of objective fn.", "");
	-- OFn;
}

void add_resource(resource **listp, resource *element)
{				/* Add an element at the front end of the list. */
	element->suc = *listp;
	*listp = element;
}

void add_oplist(oplist **listp, oplist *element)
{				/* Add an element at the front end of the list. */
	element->suc = *listp;
	*listp = element;
}

void add_product (product **listp, product *element)
{				/* Add an element at the front end of the list. */
	element->suc = *listp;
	*listp = element;
}

void add_ressch (ressch **listp, ressch *element)
{				/* Add an element at the front end of the list. */
	element->suc = *listp;
	*listp = element;
}

void add_opseq (opseq **listp, opseq *element)
{				/* Add an element at the front end of the list. */
	element->suc = *listp;
	if (*listp != NULL) (*listp)->pre = element;
	element->pre = NULL;
	*listp = element;
}

void adde_operation (operation **listp, operation *element)
{				/* Add an element at the end of the list. */
	element->suc = NULL;
	if (*listp == NULL)  {
		*listp = element;
		return;
	}

	for (; (*listp)->suc ; listp = &((*listp)->suc))
		;
	(*listp)->suc = element;
	element->pre = *listp;
	return;
}

void adde_opseq (opseq **listp, opseq *element)
{				/* Add an element at the end of the list. */
	element->suc = NULL;
	if (*listp == NULL)  {
		*listp = element;
		return;
	}

	for (; (*listp)->suc ; listp = &((*listp)->suc))
		;
```

```
        (*listp)->suc = element;
        element->pre = *listp;
        return;
}

void adde_box (box **listp, box *element)
{                          /* Add an element at the end of the list. */
        element->suc = NULL;
        if (*listp == NULL)  {
                *listp = element;
                return;
        }

        for (; (*listp)->suc ; listp = &((*listp)->suc))
                ;
        (*listp)->suc = element;
        element->pre = *listp;
        return;
}

opseq *search_op_in_opseq (operation *key, opseq *list)
{                          /* Search key in the list. */
        opseq *l;
        for (l = list ; l ; l = l->suc)
                if ( key == l->Operation ) return l;
        return NULL;
}

opseq *search_opseq (opseq *key, opseq *list)
{                          /* Search key in the list. */
        opseq *l;
        for (l = list ; l ; l = l->suc)
                if ( key->Operation == l->Operation ) return l;
        return NULL;
}

box *search_box (oplist *key, box *list)
{                          /* Search key in the list. */
        box *l;
        for (l = list ; l ; l = l->suc)
                if (key->Operation == l->Operation) return l;
        return NULL;
}

operation *search_operation (int key, operation *list)
{                          /* Search key in the list. */
        operation *l;
        for (l = list ; l ; l = l->suc)
                if ( key == l->OperationNo ) return l;
        return NULL;
}

box *search_op_in_box (operation *key, box *list)
{                          /* Search key in the list. */
        box *l;
        for (l = list ; l ; l = l->suc)
                if (key == l->Operation) return l;
        return NULL;
}

product *search_product (char *key, product *list)
{                          /* Search key in the list. */
        product *l;
        for (l = list ; l ; l = l->suc)
```

```
            if ( strcmp(key, l->ProductId) == 0 ) return 1;
      return NULL;
}

int delete_ressch (ressch **listp, ressch *element)
{                        /* Delete an element from the list. */
      ressch *temp;
      for(;*listp; listp = & ((*listp)->suc))
            if (*listp == element)  {
                  temp = (*listp)->suc;
                  free(*listp);
                  *listp = temp;
                  return 1;
            }
      return 0;
}

int delete_box (box **listp, box *element)
{                        /* Delete an element from the list. */
      box *temp;
      for(;*listp; listp = & ((*listp)->suc))
            if (*listp == element)  {
                  temp = (*listp)->suc;
                  if (temp) temp->pre = (*listp)->pre;
                  free(*listp);
                  *listp = temp;
                  return 1;
            }
      return 0;
}

int delete_opseq (opseq **listp, opseq *element)
{                        /* Delete an element from the list. */
      opseq *temp;
      for(;*listp; listp = & ((*listp)->suc))
            if (*listp == element)  {
                  temp = (*listp)->suc;
                  if (temp) temp->pre = (*listp)->pre;
                  free(*listp);
                  *listp = temp;
                  return 1;
            }
      return 0;
}


void insert_box (box *link, box *element, int pos)
{              /* Insert an element in the list at position pos relative to link.
                  pos == 1 -> after; pos == 2 -> before. */
      iif (pos == 2)  {
            if (link->pre != NULL) link->pre->suc = element;
            element->pre = link->pre;
            link->pre = element;
            element->suc = link;
            return;
      }

      element->suc = link->suc;
      link->suc = element;
      element->pre = link;
      if (element->suc != NULL)  element->suc->pre = element;
      return;
}
```

```
void error( char *a, char *b )
{               /* Print error messages and exit. */
      fprintf(stderr, "\n%s%s\n", a,b);
      exit(1);
}

void CreateFirstGeneration()
{
      int i;

      for (i=0; i<Pop ; ++i)  {
             schcpy (&Sch [i], RSch);  /* Copying RSch into Sch [i] */
             CreateAssgnmtSeq(i);
             AssgnTheSeq(i);
      }
}

void EvaluatePopulation()
{
      float sum = 0, max = 0, min = 99999, val;
      int i, size;

      size = (Generation == 0) ? Pop : TotalPopSize;
      for (i=0; i<size ; ++i)  {
             if (OFn < 0.05) val = Makespan(Sch [i]);
             else if (OFn > 0.95) val = WtdCompletion(Sch[i]);
             else
             val = Makespan(Sch [i]) * (1-OFn) + WtdCompletion(Sch [i]) *
                    OFn;

             Sch[i].Obj = val;

             sum += val;
             if (val < min)  {
                    BestSchd = i;
                    min = val;
             }
      }
      BestLast = Best;
      Best = min;
      AvgLast = Avg;
      Avg = sum/size;

      printf("Generation: %d\tAverage: %.2f\tBest: %.2f\n" , ++Generation, Avg,
      Best);
      printf("Makespan: %f, Wtd Completion: %f\n", Makespan(Sch [BestSchd]),
      WtdCompletion(Sch [BestSchd]));
}

void OutputSchedules()
{
      FILE *fout;
      int s;

      fout = fopen("BestSch", "w");
      WriteSch(0, fout);
      fclose(fout);

      printf("\nWhich schedule do you want to see in the form\n"
             "of a Gantt chart ? (-1 for none) ");
      scanf("%d", &s);
      while (s>=0 && s<TotalPopSize)  {
             gantt(Sch[0], s);
             printf("\nWhich schedule do you want to see in the form\n"
```

```
                "of a Gantt chart ? (-1 for none) ");
          scanf("%d", &s);
     }

}



void Selection()
{
     int Selected=0, j;

     /* Select elite schedules in the last generation  */
     for (Selected=0 ; Selected + EPS < elite ; ++ Selected)
          for (j=Selected+1 ; j < TotalPopSize ; ++ j)
               if (Sch [Selected].Obj > Sch [j].Obj)
                    swap (&Sch [Selected], &Sch [j]);

     while (Selected < Pop)  {
          OneSelection(&Selected);
          if ((float) random(100) <= prob_mutation)   /* Applying Mutation. */
               Mutation (Selected-1);
     }

}

void Crossover()
{
     int i, Ran;

     TotalPopSize = Pop;
     for ( i = Pop - 1 ; i > 0 ; i -= 2 )  {
          if ((float) random (100) > prob_xover) continue;
          Ran = random (i + 1) - 1;
          swap (&Sch[i], &Sch[Ran]);

          Ran = random(i) - 1;
          swap (&Sch[i-1], &Sch[Ran]);

          if (random(100) > x_type)  {
               xover(i-1,i, &TotalPopSize);
               xover(i,i-1, &TotalPopSize);
          }
          else  {
               twoptxover(i-1,i, &TotalPopSize);
               twoptxover(i,i-1, &TotalPopSize);
          }
     }
}

void xover(int a, int b, int *n)
{            /* Carry out one single point crossover operation. */
     int Ran, i;
     opseq *OpSeqp, *OpSeqp1;

     deletesch (*n);
     schcpy (& Sch[*n], RSch);
     Ran = random(NoOperations);
     for (i=0, OpSeqp = Sch[a].AssgnmtSeq; i<Ran; ++i,
                                   OpSeqp = OpSeqp->suc)  {
          OpSeqp1 = (opseq *) calloc(1, sizeof(opseq));
          OpSeqp1->Operation = OpSeqp->Operation;
```

```
            adde_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
    }

    for (OpSeqp = Sch[b].AssgnmtSeq; OpSeqp; OpSeqp = OpSeqp->suc)
        if (!search_opseq (OpSeqp, Sch[*n].AssgnmtSeq))  {
            OpSeqp1 = (opseq *) calloc(1, sizeof(opseq));
            OpSeqp1->Operation = OpSeqp->Operation;
            adde_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
        }
    AssgnTheSeq(*n);

    ++ *n;

    deletesch (*n);
    schcpy (& Sch[*n], RSch);
    for (OpSeqp = Sch[b].AssgnmtSeq; OpSeqp->suc; OpSeqp = OpSeqp->suc)
        ;
    for (i=0; i<Ran; ++i, OpSeqp = OpSeqp->pre)  {
        OpSeqp1 = (opseq * ) calloc(1, sizeof(opseq));
        OpSeqp1->Operation = OpSeqp->Operation;
        add_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
    }

    for (OpSeqp = Sch[a].AssgnmtSeq; OpSeqp->suc; OpSeqp = OpSeqp->suc)
        ;

    for (;OpSeqp; OpSeqp = OpSeqp->pre)
        if (!search_opseq(OpSeqp, Sch[*n].AssgnmtSeq))  {
            OpSeqp1 = (opseq * ) calloc(1, sizeof(opseq));
            OpSeqp1->Operation = OpSeqp->Operation;
            add_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
        }
    AssgnTheSeq(*n);
    ++ *n;
}

void twoptxover(int a, int b, int *n)
{           /* Carrying out one two point crossover. */
    int Ran1, Ran2, i;
    opseq *OpSeqp, *OpSeqp1, *OpSeqp2;

    deletesch (*n);
    schcpy (& Sch[*n], RSch);
    Ran1 = random(NoOperations-2);
    Ran2 = random(NoOperations-Ran1) + Ran1;

    for (i=0, OpSeqp = Sch[a].AssgnmtSeq; i<Ran1; ++i,
                                    OpSeqp = OpSeqp->suc)  {
        OpSeqp1 = (opseq *) calloc(1, sizeof(opseq));
        OpSeqp1->Operation = OpSeqp->Operation;
        adde_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
    }

    for (OpSeqp2 = Sch[b].AssgnmtSeq; i<Ran2; OpSeqp2 = OpSeqp2->suc)
        if (!search_opseq (OpSeqp2, Sch[*n].AssgnmtSeq))  {
            OpSeqp1 = (opseq *) calloc(1, sizeof(opseq));
            OpSeqp1->Operation = OpSeqp2->Operation;
            adde_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
            ++ i;
        }

    for (;OpSeqp ; OpSeqp = OpSeqp->suc)
        if (!search_opseq (OpSeqp, Sch [*n].AssgnmtSeq))  {
            OpSeqp1 = ( opseq * ) calloc(1, sizeof(opseq));
```

```
                OpSeqp1->Operation = OpSeqp->Operation;
                adde_opseq (& Sch [*n].AssgnmtSeq, OpSeqp1);
            }
      AssgnTheSeq(*n);

      ++ *n;

      deletesch (*n);
      schcpy (& Sch[*n], RSch);
      for (OpSeqp = Sch[a].AssgnmtSeq; OpSeqp->suc; OpSeqp = OpSeqp->suc)
            ;
      for (i=0; i<Ran1; ++i, OpSeqp = OpSeqp->pre)  {
            OpSeqp1 = (opseq *) calloc(1, sizeof(opseq));
            OpSeqp1->Operation = OpSeqp->Operation;
            add_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
      }

      for (OpSeqp2 = Sch[b].AssgnmtSeq; OpSeqp2->suc; OpSeqp2 = OpSeqp2->suc)
            ;

      for (; i<Ran2 ; OpSeqp2 = OpSeqp2->pre)
            if (!search_opseq(OpSeqp2, Sch[*n].AssgnmtSeq))  {
                  OpSeqp1 = (opseq * ) calloc(1, sizeof(opseq));
                  OpSeqp1->Operation = OpSeqp2->Operation;
                  add_opseq (&Sch[*n].AssgnmtSeq, OpSeqp1);
                  ++ i;
            }

      for (; OpSeqp ; OpSeqp = OpSeqp->pre)
            if (!search_opseq (OpSeqp, Sch [*n].AssgnmtSeq))  {
                  OpSeqp1 = (opseq *) calloc(1, sizeof(opseq));
                  OpSeqp1->Operation = OpSeqp->Operation;
                  add_opseq (& Sch [*n].AssgnmtSeq, OpSeqp1);
            }

      AssgnTheSeq(*n);
      ++ *n;
}

void OneSelection(int *s)
{
      int Ran;

      Ran = random(TotalPopSize - *s) + *s -1;
      swap (&Sch [Ran], &Sch [*s]);

      Ran = random(TotalPopSize - *s - 1) + *s;
      if (Sch [Ran].Obj < Sch [*s].Obj) swap (&Sch [Ran], &Sch [*s]);

      ++ *s;
}

void CreateAssgnmtSeq(int p)
{
      opseq *SchedulableOps = NULL, *OpSeqp1, *OpSeqp, *OpSeqp2;
      int NoSchedulables = 0, i, Ran;
      oplist *Oplstp;
      product *Prodp;

      for (Prodp = ProdList; Prodp ; Prodp = Prodp->suc)
            if ( schedulable( Prodp->First, p ) )  {
                  ++ NoSchedulables;
                  OpSeqp = ( opseq * ) calloc(1, sizeof(opseq));
                  OpSeqp->Operation = Prodp->First;
```

```
                add_opseq ( & SchedulableOps, OpSeqp);
            }
     while ( NoSchedulables )  {
          Ran = random ( NoSchedulables );
          for (i=1, OpSeqp = SchedulableOps ; i != Ran ; ++i,
                                        OpSeqp = OpSeqp->suc)
                ;

          OpSeqp2= ( opseq * ) calloc(1, sizeof(opseq));
          OpSeqp2->Operation = OpSeqp->Operation;
          adde_opseq( & Sch[p].AssgnmtSeq, OpSeqp2);
          for (Oplstp=OpSeqp->Operation->Succ ; Oplstp ; Oplstp =
                                             Oplstp->suc)
                if ( schedulable(Oplstp->Operation, p) )  {
                     OpSeqp1 = ( opseq * ) calloc(1, sizeof(opseq));
                     OpSeqp1->Operation = Oplstp->Operation;
                     add_opseq (& SchedulableOps, OpSeqp1);
                     ++ NoSchedulables;
                }

          delete_opseq (&SchedulableOps, OpSeqp);
          -- NoSchedulables;
     }
}

int schedulable ( operation *Op, int p )

/* Checks whether Op is schedulable or not by checking all of its predecessors
and by searching for its predecessor in the assignment sequence.  If all of the
predecessors have been scheduled and if they occur in the assignment sequence,
then Op is schedulable (returns 1) otherwise not (returns 0). */

{
     oplist *r;

     for (r = Op->Pred; r ; r=r->suc)  {
          if (r->Operation->Schd == 'Y') continue;
          if (!search_op_in_opseq (r->Operation, Sch[p].AssgnmtSeq))
               return 0;
     }
     return 1;
}

void AssgnTheSeq(int p)  /* Create an active schedule, using the assignment
                            sequnce. */
{
     ressch **OneTypeRes, **OneTypeResSlots, *Resschp;
     oplist *Oplstp;
     opseq *OpSeqp;
     box *Boxp, *BoxpNew, *asd;
     float EST = 0., min, ftime;
     int i, NoRessch, Ran, NoInt, found, NoSlots, *IntArray, j, k, l;
     resource *Resp, *ESTChanger;

     OneTypeRes = OneTypeResSlots = NULL;
     IntArray = NULL;

     for (OpSeqp = Sch[p].AssgnmtSeq ; OpSeqp ; OpSeqp = OpSeqp->suc)  {
          EST = -1;
          if (OpSeqp->Operation->Pred == NULL)  {
               EST = DtToTime( (search_product(OpSeqp->Operation->
                    ProductId, ProdList))->ReleaseDt,
                    Sch [p].StartingDt);
          }
```

```
    else for(Oplstp = OpSeqp->Operation->Pred; Oplstp ;
                                    Oplstp=Oplstp->suc)
        for (Resschp=Sch[p].Resources; Resschp;
                        Resschp=Resschp->suc)   {
            Boxp = search_box(Oplstp, Resschp->JobSeq);
            if (Boxp)  {
                ftime = FinishTime(Boxp);
                if (ftime > EST) EST = ftime;
                break;
            }
        }

    if (EST < 0)   {
        printf("\nIndividual: %d, Operation no.: %d", p,
                        OpSeqp->Operation->OperationNo);
        error("cannot find EST for operation on ",
                        OpSeqp->Operation->ProductId);
    }

    Resp = ESTChanger = OpSeqp->Operation->ResList;

while(1)   { /* circular loop on operation's resources (Resp) */
    NoRessch = 0;
    for(Resschp = Sch[p].Resources; Resschp ; Resschp =Resschp->suc)
        if (strcmp(Resschp->ResourceType, Resp->Resource)==0)  {
            ++ NoRessch;
            OneTypeRes =  (ressch **) realloc(OneTypeRes,
                        NoRessch*sizeof(ressch *));
            OneTypeRes[NoRessch-1] = Resschp;
        }
    if (NoRessch < Resp->No)
        error("not enough resources for:", Resp->Resource);

    while (1)   {
        NoSlots = 0;
        for (i=0 ; i<NoRessch ; ++i)    {
            for(BoxP=OneTypeRes[i]->JobSeq; Boxp &&
                        Boxp->StartTime + EPS < EST;
                        Boxp = Boxp->suc)
                ;
            if (Boxp == NULL)   {
                for(Boxp = OneTypeRes[i]->JobSeq; Boxp
                        && Boxp->suc ;
                        Boxp = Boxp->suc) ;
                if  (Boxp == NULL ||
                        FinishTime(Boxp) <= EST + EPS )
                        ++ NoSlots;
            }
            else if (Boxp->StartTime - EST + EPS >=
                        ProcessTime(OpSeqp->Operation))
                if (Boxp->pre == NULL ||
                FinishTime(Boxp->pre) <= EST + EPS)
                ++ NoSlots;
        }
        if (NoSlots >= Resp->No) break;

        min = 99999;

        for (i=0; i<NoRessch ; ++i)  {
            for (Boxp=OneTypeRes[i]->JobSeq; Boxp &&
                        (ftime = FinishTime(Boxp)) <
                        EST + EPS ; Boxp = Boxp->suc)
                ;
            if (Boxp != NULL && ftime < min) min = ftime;
```

```
            }
            EST = min;
            ESTChanger = Resp;
        }
    -   Resp = (Resp->suc) ? Resp->suc : OpSeqp->Operation->ResList;
        if (ESTChanger == Resp) break;
}                   /* End of the circular loop on Resp */

for (Resp = OpSeqp->Operation->ResList; Resp; Resp = Resp->suc)   {
        NoRessch = 0;
        for (Resschp = Sch[p].Resources; Resschp; Resschp =
                                          Resschp->suc)
            if (strcmp(Resschp->ResourceType, Resp->Resource)==0)
            {
                ++ NoRessch;
                OneTypeRes = (ressch **) realloc(OneTypeRes,
                        NoRessch * sizeof(ressch *));
                OneTypeRes[NoRessch-1] = Resschp;
            }
        NoSlots = 0;
        for (i=0; i<NoRessch ; ++i)   {
            for( Boxp = OneTypeRes[i]->JobSeq; Boxp &&
                    Boxp->StartTime < EST + EPS ; Boxp =
                    Boxp->suc)
                ;
            if (Boxp == NULL)   {
                for(Boxp = OneTypeRes[i]->JobSeq; Boxp
                        && Boxp->suc ;
                        Boxp = Boxp->suc) ;
                if  (Boxp == NULL ||
                    FinishTime(Boxp) <= EST + EPS )   {
                    ++ NoSlots; /* Found a slot */
                    OneTypeResSlots = (ressch **)
                            realloc(OneTypeResSlots,
                            NoSlots *
                            sizeof(ressch *));
                    OneTypeResSlots[NoSlots-1] =
                                OneTypeRes[i];
                }
            }
            else if (Boxp->StartTime - EST + EPS >=
                    ProcessTime(OpSeqp->Operation))
                if (Boxp->pre == NULL ||
                FinishTime(Boxp->pre) <= EST + EPS)  {

                ++ NoSlots; /* Found a slot */
                OneTypeResSlots = (ressch **)
                            realloc(OneTypeResSlots,
                            NoSlots *
                            sizeof(ressch *));
                OneTypeResSlots[NoSlots-1]=OneTypeRes[i];
            }
        }
        NoInt = 0;
        for(i=0; i<Resp->No ; ++i)   {
            Ran = random(NoSlots --);
            j=0;
            for (k=0 ; j != Ran ; ++k)   {
                found = 0;
                for(l=0; l<NoInt ; ++l)
                    if (k == IntArray[l])   {
                        found=1;
                        break;
                    }
```

```
                    if (found == 0) ++j;
                }
                --k;  /* Do the scheduling on the kth Slot */
                ++ NoInt;
                IntArray = (int *) realloc(IntArray, NoInt *
                                         sizeof(int));
                IntArray[NoInt-1] = k;
                for (Boxp = OneTypeResSlots[k]->JobSeq; Boxp &&
                        Boxp->StartTime < EST + EPS ;
                        Boxp=Boxp->suc)
                    ;

                BoxpNew = (box * ) calloc(1, sizeof( box ));
                BoxpNew->StartTime = EST;
                BoxpNew->Operation = OpSeqp->Operation;

                if (Boxp != NULL)  {
                    if (Boxp->pre == NULL)
                        OneTypeResSlots[k]->JobSeq = BoxpNew;
                    insert_box (Boxp, BoxpNew, 2);
                }
                else  {
                    for(Boxp = OneTypeResSlots[k]->JobSeq ; Boxp &&
                            Boxp->suc; Boxp = Boxp->suc)
                        ;
                    if (Boxp != NULL)
                        insert_box (Boxp, BoxpNew, 1);
                    else
                        OneTypeResSlots[k]->JobSeq = BoxpNew;
                }
            }
        }
        }
}

float WtdCompletion(schedule s)
{           /* Calculate Weighted completion of schedule s */
        float Obj = 0;
        ressch *Resschp;
        box *Boxp;
        product *Prodp;
        int p=0;
        operation *Opp;

        for (Prodp = ProdList ; Prodp ; Prodp = Prodp->suc)  {
            for (Opp = Prodp->First ; Opp && Opp->suc ; Opp = Opp->suc)
                ;
            if (Opp == NULL)  continue;
            for (Resschp=s.Resources; Resschp; Resschp=Resschp->suc) {
                Boxp = search_op_in_box(Opp, Resschp->JobSeq);
                if (Boxp)  {
                    Obj += Prodp->Priority * FinishTime(Boxp);
                    p += Prodp->Priority;
                    break;
                }
            }
        }
        return Obj/p;
}

float Makespan(schedule a)
{               /* Calculate makespan of schedule a. */
        float f, Obj = 0;
        ressch *Resschp;
```

```
	box *Boxp;

	for (Resschp = a.Resources; Resschp ; Resschp = Resschp->suc)  {
		for (Boxp=Resschp->JobSeq; Boxp && Boxp->suc ; Boxp = Boxp->suc)
	-		;
		if (Boxp)  {
			f = FinishTime(Boxp);
			if (f > Obj) Obj = f;
		}
	}

	return Obj;
}

float TimeSince1996Till (date Dt)
{		/* Calculates time elapsed since 1st Jan 1996 till Dt. */
	int i, l, days = 0;

	for (i=1996; i<Dt.Year; ++i)
		days += NoOfDays(i);

	l = leap(Dt.Year);

	for(i=1; i<Dt.Month ; ++i)
		days += daytab[l][i];

	days += Dt.Day;

	return( DayToTime (days) );
}

float DayToTime (int day)
{	/* Converts no. of days to time.  Here both are same. */
	return( (float) day );
}

int leap (int y)
{	/* Returns the condition (1 when true, 0 when false) of leap year. */
	return( y%4 == 0 && y%100 != 0 || y%400 == 0);
}

int TimeToDay (float t)
{	/* Converts time to no. of days. */
	int T;

	T = (int) t;
	return( (t-T >= 0.5) ? T+1 : T);
}

float DtToTime( date Rel, date RSchDt )
{	/* Converts the difference of dates to time. */
	return(TimeSince1996Till (Rel) - TimeSince1996Till (RSchDt));
}

date TimeToDt (float t)
/* Converts time to date.  Useful when printing the Gantt chart in terms
of dates rather than in terms of time. */
{
	int days, i, l;
	date dt;

	days = TimeToDay(t);

	for (i=1996; days > NoOfDays(i) ; ++i)
```

```
            days -= NoOfDays(i);

        dt.Year = i;
        l = leap(i);

        for(i=1; days>daytab[l][i] ; ++i)
              days -= daytab[l][i];

        dt.Month = i;
        dt.Day = days;

        return dt;
}

int NoOfDays(int y)
/* Calculates no. of days in a year. */
{
        return (365 + leap(y));
}

void WriteSch (int i, FILE *f)
/* Writes schedule in the same format as the Resource schedule.  Useful
when a system is built in which this program is implemented.  Other
supporting programs can read this format using a function same as
Read_ResourceFile(). */
{
        box *Boxp;
        ressch *Resschp;

        fprintf(f, "Date: %d %d %d", Sch[i].StartingDt.Day,
                    Sch[i].StartingDt.Month, Sch[i].StartingDt.Year);

        for (Resschp= Sch[i].Resources ; Resschp ; Resschp = Resschp->suc)   {
              fprintf(f, "\n\nResource_type: %s\nResource_id: %s\n",
                          Resschp->ResourceType, Resschp->ResourceId);

              for (Boxp=Resschp->JobSeq; Boxp; Boxp=Boxp->suc)   {
                    fprintf(f, "\n\tProduct_id: ");
                    if (strncmp(Boxp->Operation->ProductId, "Down", 4) == 0)
                          fprintf(f, "Down\n\tComment: %s\n\tStart_time: "
                                "%f\n\tDown_time: %f\n",
                                Boxp->Operation->ProductId + 4,
                                Boxp->StartTime,
                                Boxp->Operation->ProcTime);
                    else  fprintf(f, "%s\n\tOperation_no.: %d\n\t"
                          "Start_time: %f\n", Boxp->Operation->ProductId,
                          Boxp->Operation->OperationNo, Boxp->StartTime);
              }
        }
        fprintf(f, "\n\n****\n\nObjective Function value = %f\n", Sch[i].Obj);
}

int random (int l)
/* Generates a random integer between 1 to l. */
{
        int Ran, seed;
        static int firsttime = 1;

        if (firsttime)   {
              seed = time(NULL)%1000;
              srand(seed);
              firsttime = 0;
        }
```

```
	Ran = (( (float) rand() )/RAND_MAX) * 1;
	if (++ Ran > 1)  Ran = 1;
	return Ran;
}

int schcmp(const schedule *A, const schedule *B)
/* Compares two schedules A and B on the basis of their objective
functions.  Used in sorting the schedules by qsort() before final
output for selecting the best schedule. */
{
	if (A->Obj == B->Obj) return 0;
	return( (A->Obj < B->Obj) ? -1: 1 );
}

void schcpy( schedule *To, const schedule *From)
/* Copies schedule From to To. */
{
	box *Boxp, *boxtemp;
	ressch *Resschp, *restemp;
	opseq *OpSeqp, *seqtemp;

	To->StartingDt = From->StartingDt;
	To->Obj = From->Obj;

	for ( Resschp = From->Resources; Resschp ; Resschp=Resschp->suc)  {
		restemp = (ressch * ) calloc(1, sizeof(ressch));
		strcpy(restemp->ResourceType, Resschp->ResourceType);
		strcpy(restemp->ResourceId, Resschp->ResourceId);

		for (Boxp= Resschp->JobSeq; Boxp; Boxp=Boxp->suc)  {
			boxtemp = (box * ) calloc(1, sizeof( box ));
			boxtemp->StartTime = Boxp->StartTime;
			boxtemp->Operation = Boxp->Operation;
			adde_box (&restemp->JobSeq, boxtemp);
		}
		restemp->JobSeqR = boxtemp;
		add_ressch (&To->Resources, restemp);
	}
	for(OpSeqp = From->AssgnmtSeq; OpSeqp; OpSeqp = OpSeqp->suc)  {
		seqtemp = (opseq * ) calloc(1, sizeof(opseq));
		seqtemp->Operation = OpSeqp->Operation;
		adde_opseq (& To->AssgnmtSeq, seqtemp);
	}
	return;
}

float FinishTime(box *b)
/* Calculates at what time the box (an operation in the Gantt chart)
will finish. */
{
	if (b == NULL) error("no box here for FinishingTime", "");
	return(b->StartTime + b->Operation->ProcTime + b->Operation->Margin);
}

float ProcessTime( operation *p)
/* Calculates the processing time of an operation. */
{
	if (p == NULL) error("no operation here for ProcessingTime", "");
	return(p->ProcTime + p->Margin);
}

void deletesch (int a)
/* Deletes the 'a'th schedule in the population. */
{
```

```
	ressch *Resschp;

	Sch[a].StartingDt.Day = 0;
	Sch[a].StartingDt.Month = 0;
	Sch{a].StartingDt.Year = 0;
	Sch[a].Obj = 0;

	for (Resschp = Sch[a].Resources ; Resschp ; Resschp = Resschp->suc)   {
		while (delete_box (& Resschp->JobSeq, Resschp->JobSeq))
			;
		Resschp->ResourceId[0] = '\0';
		Resschp->ResourceType[0] = '\0';
	}

	while (delete_ressch (& Sch[a].Resources, Sch[a].Resources))
		;
	while (delete_opseq (& Sch[a].AssgnmtSeq, Sch[a].AssgnmtSeq))
		;

	/* Now everything is set to zero */
}

void gantt (schedule s, int i)
/* Writes schedule s in the form of a Gantt chart.  i is the suffix
to the filename "gantt". */
{
	opseq *OpSeqp;
	ressch *r;
	box *b;
	int print;
	char fout[10];
	float t_elapsed, mkspan, ti;
	FILE *f;

	sprintf (fout, "gantt%d", i);
	f = fopen(fout, "w");

	printf("\nPreparing the Gantt chart . . . ");
	fprintf(f, "Time|");
	for (r=s.Resources ; r ; r=r->suc)
		fprintf(f, "%-3.3s|", r->ResourceType);
	fprintf(f, "\n    |");
	for (r=s.Resources ; r ; r=r->suc)
		fprintf(f, "%-3.3s|", r->ResourceId);
	fprintf(f, "\n");

	mkspan = Makespan(s);

	for (t_elapsed = 0; t_elapsed <= mkspan ; t_elapsed +=unit)  {
		fprintf(f, "%4.0f|", t_elapsed);
		for (r = s.Resources ; r ; r=r->suc)  {
			print = 0;
			for ( b=r->JobSeq ; b ; b=b->suc)
				if ((b->StartTime <= t_elapsed  + EPS &&
				    FinishTime(b) > t_elapsed))  {
				      print = 1;
				      break;
				}
			if (print)
				fprintf(f, "%-3.3s|", b->Operation->ProductId);
			else
				fprintf(f, "   |");
		}
		fprintf(f, "\n    |");
```

```
            for (r = s.Resources ; r ; r=r->suc)   {
                  print = 0;
                  for ( b=r->JobSeq ; b ; b=b->suc)
                        if ((b->StartTime <= t_elapsed  + EPS &&
                            FinishTime(b) > t_elapsed))   {
                              print = 1;
                              break;
                        }
                  if (print)
                        fprintf(f, "%-3d|", b->Operation->OperationNo);
                  else
                        fprintf(f, "    |");
            }
            fprintf(f, "\n");
      }
      fprintf(f, "Time|");
      for (r=s.Resources ; r ; r=r->suc)
            fprintf(f, "%-3.3s|", r->ResourceType);
      fprintf(f, "\n    |");
      for (r=s.Resources ; r ; r=r->suc)
            fprintf(f, "%-3.3s|", r->ResourceId);
      fprintf(f, "\n");

      if (s.AssgnmtSeq)
            for (OpSeqp = s.AssgnmtSeq ; OpSeqp ; OpSeqp = OpSeqp->suc)
                  fprintf(f, "(%s,%d) ", OpSeqp->Operation->ProductId,
                              OpSeqp->Operation->OperationNo);

      fprintf(f, "\nMakespan: %f, Wtd Completion: %f\n",
      Makespan(s), WtdCompletion(s));

      fclose(f);
      printf(". . . Over!\n");
}

void swap (schedule *s1, schedule *s2)
/* Swaps schedule s1 with s2. */
{
      schedule temp;

      temp = *s1;
      *s1 = *s2;
      *s2 = temp;
}

int mutn (opseq **Seq, int N)
/* Mutates the operation sequence Seq at the 'N'th place by interchanging
two operations.  Returns 1 if the mutation is done i.e., if mutation
results in a feasible solution; otherwise 0. */
{
      opseq *OpSeqp, *OpSeqp1;
      oplist *Oplstp;
      int i;

      for (OpSeqp = *Seq, i=1; i<N ; OpSeqp = OpSeqp->suc, ++i)
            ;
      if (OpSeqp->suc == NULL) return 0;
      else OpSeqp1 = OpSeqp->suc;

      for (Oplstp=OpSeqp->Operation->Succ ; Oplstp ; Oplstp = Oplstp->suc)
            if (Oplstp->Operation == OpSeqp1->Operation)
                  return 0;

      /* Interchange OpSeqp & OpSeqp1 in the list. */
```

```
        if (OpSeqp->pre) OpSeqp->pre->suc = OpSeqp1;
        OpSeqp1->pre = OpSeqp->pre;
        OpSeqp->suc = OpSeqp1->suc;
        if (OpSeqp->suc) OpSeqp->suc->pre = OpSeqp;
        OpSeqp->pre = OpSeqp1;
        OpSeqp1->suc = OpSeqp;

        if (OpSeqp1->pre == NULL) *Seq = OpSeqp1;

        return 1;
}

int Mutation (int i)
/* Mutates the 'i'th schedule in the population. */
{
        opseq *Seq;

        Seq = Sch [i].AssgnmtSeq;
        if (mutn (&Seq, random (NoOperations)))  {
                Sch [i].AssgnmtSeq = NULL;
                deletesch (i);
                schcpy (& Sch [i], RSch);
                Sch [i].AssgnmtSeq = Seq;
                AssgnTheSeq (i);
                return 1;
        }
        return 0;
}

/* * * * * * * * * * * * * *  End of the GA program  * * * * * * * * * * * * * * */
```

FLOWCHART SHOWING THE IMPLEMENTATION OF GENETIC ALGORITHM FOR GENERALISED JOB SHOP SCHEDULING

START

CREATE RANDOM SOLUTIONS FOR **FIRST GENERATION.**
For each individual:
- Create a new random assignment sequence.
- Allocate operations to the resources based on the assignmrnt sequence

`CreateFirstGeneration()`

EVALUATE THE POPULATION
- Calculate objective function value of each individual
- Print the best value (minimum) and the average objective function value.

`EvaluatePopulation()`

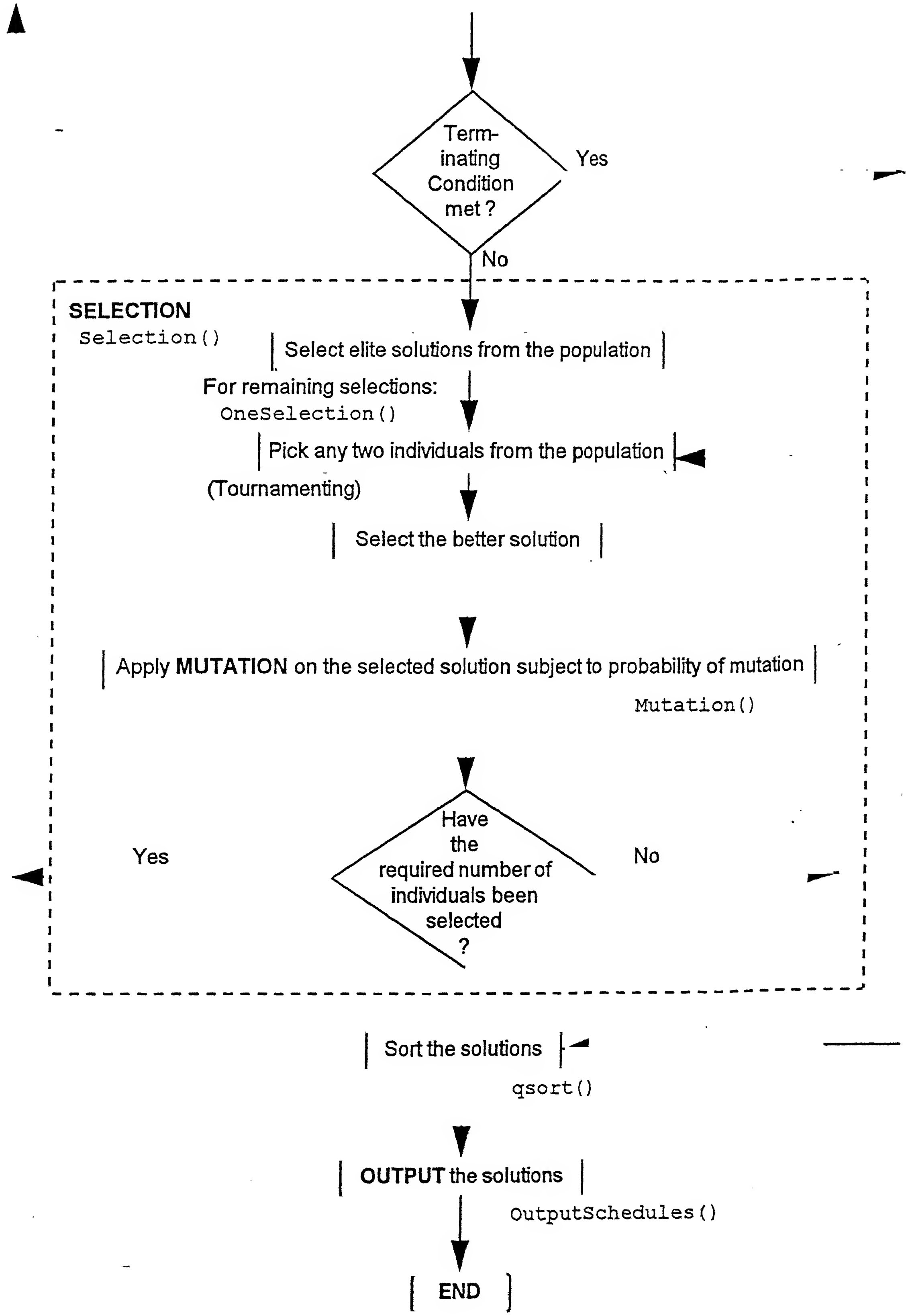
APPLY **CROSSOVER** OPERATOR
For all individuals:
- Select any two random individuals.
- Crossover them subject to crossover probability.

`Crossover()`

EVALUATE THE POPULATION
- Calculate objective function value of each individual.
- Print the best value (minimum) and the average value.

`EvaluatePopulation()`

EVALUATE THE **TERMINATING CONDITION**
TerminatingCond = Generation > NGen



Note: Only those crossover operators (1 point crossover, 2 point crossover) and mutation operator (Adjacent two-job interchange) were selected which ensure feasibility of solutions.

A 121769

Date Slip

This book is to be returned on the date last stamped.